1ª ESCOLA DE INFORMÁTICA TEÓRICA E MÉTODOS FORMAIS
22–23 de novembro de 2016
Natal – RN

ANAIS

Editora
Sociedade Brasileira de Computação – SBC

Organizadores
Simone André da Costa Cavalheiro
Martin Alejandro Musicante
Leila Ribeiro
Marcel Vinicius Medeiros Oliveira

Realização
Sociedade Brasileira de Computação
Universidade Federal do Rio Grande do Norte
Universidade Federal de Pelotas

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

(Biblioteca do Instituto de Informática da UFRGS, Porto Alegre, RS)

Escola de Informática Teórica e Métodos Formais (2016 nov 22–23: Natal)

Anais — Natal: SBC / UFPel / UFRN, 2016.

182 p.: il.

ISBN 978-85-7669-357-4

1. Métodos Formais. I. Cavalheiro, Simone André da Costa. II. Musicante, Martin Alejandro. III. Ribeiro, Leila. IV. Oliveira , Marcel Vinicius Medeiros. V. Título.

É proibida a reprodução total ou parcial desta obra sem o consentimento prévio dos autores

ETMF 2016

http://etmf2016.imd.ufrn.br

Comitê de Programa

Aline Maria Santos Andrade (UFBA)
Ana Cristina Vieira de Melo (USP)
Anamaria Martins Moreira (UFRJ)
Arnaldo Vieira Moura (UNICAMP)
Benjamín René Callejas Bedregal (UFRN)
Breno Piva Ribeiro (UFS)
Carlos Alberto Olarte Vega (UFRN)
Christiano Braga (UFF)
Cláudia Nalon (UnB)
Giovanny Fernando Lucero Palma (UFS)
Jayme Szwarcfiter (UFRJ)
Joao Marcos (UFRN)
Juliana Kaizer Vizzotto (UFSM)
Juliano Manabu Iyoda (UFPE)
Leila Ribeiro (UFRGS)
Leila Maciel de Almeida e Silva (UFS)
Luciana Foss (UFPel)
Lucio Mauro Duarte (UFRGS)
Marcel Vinicius Medeiros Oliveira (UFRN)
Marcelo de Almeida Maia (UFU)
Marcio Lopes Cornélio (UFPE)
Martin Alejandro Musicante (UFRN)
Patricia Duarte de Lima Machado (UFCG)
Regivan Hugo Nunes Santiago (UFRN)
Renata Hax Sander Reiser (UFPel)
Rohit Gheyi (UFCG)
Rosiane de Freitas Rodrigues (UFAM)
Sérgio Queiroz de Medeiros (UFRN)
Simone André da Costa Cavalheiro (UFPel)
Tiago Lima Massoni (UFCG)
Umberto Souza da Costa (UFRN)

Revisores Adicionais

Bruno Lopes (UFF)
Camilo Rueda (Pontificia Universidad Javeriana-Cali, Colombia)
Franklin de Lima Marquezino (UFRJ)
João Batista de Souza Neto (doutorando UFRN)
Paulo Eustáquio Duarte Pinto (UERJ)
Ranieri Batista Costa (doutorando PUC-Rio)

Comitê Organizador

Coordenação Geral
Profa. Dra. Simone André da Costa Cavalheiro (UFPel)

Organização Local
Prof. Dr. Martin Alejandro Musicante (UFRN)

Organização do Comitê de Programa
Profa. Dra. Simone André da Costa Cavalheiro (UFPel)

Comissão de Organização
Profa. Dra. Simone André da Costa Cavalheiro (UFPel)
Prof. Dr. Martin Alejandro Musicante (UFRN)
Profa. Dra. Leila Ribeiro (UFRGS)
Prof. Dr. Marcel Vinicius Medeiros Oliveira (UFRN)

Apresentação

Com imensa satisfação apresentamos a 1^a Escola de Informática Teórica e Métodos Formais (ETMF 2016). A ETMF 2016 é uma promoção conjunta da Universidade Federal de Pelotas (UFPel) e Universidade Federal do Rio Grande do Norte (UFRN), ocorrendo 22 e 23 de novembro, em Natal, Rio Grande do Norte. Esta é a primeira edição da escola que tem por objetivo congregar estudantes e pesquisadores para divulgar e promover aspectos teóricos da Computação. O evento conta com tutoriais, minicursos e sessões técnicas discutindo temas atuais e relevantes da área.

Os tutoriais e minicursos objetivam qualificar a formação de estudantes e profissionais nas áreas que compõem a informática teórica. Nas sessões técnicas são apresentados trabalhos concluídos ou em andamento, relacionados com pesquisas na área. Os artigos são inicialmente revisados em um processo onde pelo menos dois revisores avaliaram cada artigo, a fim de garantir a qualidade e, ao mesmo tempo, apresentar aos autores sugestões relevantes para seus trabalhos. Desta forma, agradecemos ao Comitê de Programa e revisores pelo excelente trabalho na seleção dos textos que compõem este livro. Nesta Escola, 17 artigos de um total de 22 foram aceitos para publicação e apresentação.

Finalmente, agradecemos aos tutorialistas e ministrantes de minicursos pela aceitação do convite para fazerem suas apresentações e aos autores pelo interesse em submeter seus trabalhos à escola. Também agradecemos o suporte da ClearSy, do Instituto Metrópole Digital (IMD) e da UFRN, os quais acreditaram no evento e o viabilizaram. Estendemos nossa saudação à SBC, ao DIMAp UFRN, à UFPel e à UFRGS pelo apoio para a realização deste evento.

Desejamos a todos os participantes que aproveitem bem a estadia em Natal e que tenham uma excelente Escola.

Simone André da Costa Cavalheiro
Martin Alejandro Musicante
Leila Ribeiro
Marcel Vinicius Medeiros Oliveira

Natal, novembro de 2016.

Sumário

ETMF 2016 • 22–23 de novembro de 2016 • ISBN 978-85-7669-357-4

Small Normal Form for Propositional Logic: Dynamic Programming Approach

Cláudia Nalon[1] and Matheus C. S. C. Pimenta[1]

Department of Computer Science, University of Brasília
C.P. 4466 – CEP:70.910-090 – Brasília – DF – Brazil
nalon@unb.br, matheuscscp@gmail.com

Abstract. The satisfiability problem for the classical propositional logic is intractable (unless P = NP). In practice, however, preprocessing might allow for the simplification of the input formula, thus improving the efficiency of automated tools for checking satisfiability. Many of the proof methods for propositional logic employ normal forms. For instance, SAT-based and resolution-based methods are applied to formulae into Conjunctive Normal Form (CNF), that is, formulae are conjunctions of clauses. In this paper, we describe a renaming procedure, based on dynamic programming, for reducing the number of clauses generated by the transformation of a formula into its CNF. We follow previous approaches, by avoiding renaming of formulae, if the usual distribution rewriting rule generates less clauses. Our procedure is correct, but it is not optimal, if we consider the class of linear formulae (i.e. without equivalences and repeated formulae). However, experimental evaluation shows that our procedure is competitive when formulae are not linear.

1 Introduction

Propositional Logic, PL, despite being one of the simplest logical formalisms around, is sufficiently expressive to describe several interesting problems in Computer Science: any problem in NP can be formalised as an instance of the satisfiabiality problem of PL [3]. Practical applications include, for instance, hardware synthesis, optimisation and verification [1,11,7], and applications in biological and medical sciences [8]. Problems in NP are considered intractable, i.e. it is not known if there is a polynomial-time bounded algorithm that can decide on these problems. In practice, however, preprocessing might improve on the efficiency of existing satisfiability methods for PL.

There is a multitude of computational methods for dealing with the satisfiability problem for propositional logic. We are particularly interested in resolution-based methods [16], as the present study is a first step towards the implementation of efficient preprocessing techniques, which are going to be added to an existing hyperresolution-based [15] theorem prover for propositional modal logic [10]. For the propositional case, the resolution method consists of only one inference rule, which is exhaustively applied to a set of clauses until either the empty clause is generated (in which case the set is unsatisfiable) or no new clauses can be generated (in which case the set is satisfiable). Thus,

in order to use the resolution method for testing the satisfiability of a propositional formula, the first step is the transformation of such formula into a (equisatisfiable) set of clauses or, equivalently, into its Conjunctive Normal Form (CNF).

The CNF of a formula can be obtained by applying well-known rewriting rules. This transformation ensures that the transformed formula is semantically equivalent to the original one. However, one of the rewriting rules used in this transformation, namely the distribution rule (from $\varphi \vee (\psi \wedge \chi)$ obtain $(\varphi \vee \psi) \wedge (\varphi \vee \chi)$), leads to an exponential blow-up in the size of the formula. In order to avoid this problem, another technique known as *renaming* [17,13], which results in a formula linearly bounded in the size of the original formula, is used instead. Renaming consists of introducing new propositional symbols p for each subformula ψ occurring in the original formula φ and adding the *definition*, $p \Leftrightarrow \psi$, to the formula. Formally, let $\mathsf{repb}(\varphi, \psi, \chi)$ denote the replacement of every occurrence of ψ in φ by χ. The renaming of ψ in φ is given by $\mathsf{repb}(\varphi, \psi, p) \wedge (p \Leftrightarrow \psi)$, where p does not occur in φ. In the case where ψ occurs only with positive polarity, the definition simplifies to the implication $p \Rightarrow \psi$. The renaming technique preserves satisfiability: the resulting formula is satisfiable if, and only if, the original one is satisfiable [17,13].

It is tempting to think that all is sorted out at this point, but that is not quite the case. The complexity of resolution based-methods, as well as the complexity of other satisfiability methods for propositional logics (e.g. DPLL [6,5]), is deterministic exponential in the number of propositional symbols in the input formula. Although the renaming technique introduces a linear number of such symbols, thus the complexity for the transformed problem is asymptotically the same as that of the original one, in practice is desirable to avoid the introduction of new symbols and their definitions whenever the size of the formula does not increase if the usual rewriting rules are applied. For instance, assume that $\varphi = p \vee (q \wedge r \wedge s)$ is the input formula. Applying distribution to φ results in $(p \vee q) \wedge (p \vee r) \wedge (p \vee s)$. Using the renaming technique, the subformula $(q \wedge r \wedge s)$ in the input formula is replaced by t, a new propositional symbol, and the definition $t \Rightarrow (q \wedge r \wedge s)$ must be added to the formula. The resulting CNF is $(p \vee t) \wedge (\neg t \vee q) \wedge (\neg t \vee r) \wedge (\neg t \vee s)$, which has more propositional symbols and is bigger than the formula obtained by the application of distribution.

In [2], a top-down renaming technique which tries to avoid the situation of the previous example by carefully deciding whether to apply either distribution or renaming during the transformation of a formula into its CNF, is given. The optimality criteria is based on the number of clauses, which seems to be a reasonable criteria in practice [12]. In [2], top-down renamings are shown to be optimal with respect to the number of clauses for the class of linear formulae (i.e. when a formula does not contain equivalences and repeated formulae). The greedy algorithm proposed in [2] is shown to produce a top-down renaming, therefore it is optimal for linear formulae. We follow this approach by presenting an algorithm based on dynamic programming. We can show that our approach is not optimal with respect to the proposed criteria, but experimental evaluation of both techniques show that our approach may produce less clauses when the input formula is not linear and may be chosen as good heuristic to deal with the transformation of formulae into CNF for unrestricted classes of formulae.

The paper is organised as follows. We introduce the syntax and semantics of PL in Section 2. We briefly review the techniques for obtaining the CNF of a formula in Section 3. Section 4 describes our implementation and discuss our results. We summarise our results in Section 5.

2 Language

We first define the syntax of PL.

Definition 1. *Let* $\mathsf{P} = \{p, q, \ldots, t, \ldots, p', q', \ldots\}$ *be a denumerable set of propositional symbols. The* set of well-formed formulae*,* WFF*, is the least set such that every* $p \in \mathsf{P}$ *is in* WFF*; if* φ *and* φ_i *are in* WFF*, then so are* $\neg\varphi$ *and* $\wedge_{i=1}^{n}\varphi_i$*, for* $n \in \mathbb{N}$*.*

The formulae **false**, **true**, $(\varphi \vee \psi)$, $(\varphi \Rightarrow \psi)$, and $(\varphi \Leftrightarrow \psi)$ are introduced as the usual abbreviations for $(\varphi \wedge \neg\varphi)$, $\neg$**false**, $\neg(\wedge_{i=1}^{n}\neg\varphi_i)$, $(\neg\varphi \vee \psi)$, and $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$, respectively (where $\varphi, \varphi_i, \psi \in \mathsf{WFF}, n \in \mathbb{N}$).

A *literal* is either a propositional symbol or its negation. A clause is a disjunction of literals. The notions of subfomulae and proper subformulae are defined as usual. We denote by $\varphi \sqsubset \varphi'$ and $\varphi \sqsubseteq \varphi'$ that φ is a subformula and proper subformula of φ', respectively. The size of a formula is also defined in the usual way:

Definition 2. *Let* $\varphi \in \mathsf{WFF}$*. The size of* φ*, denoted by* $|\varphi|$*, is given as follows. If* $\varphi \in \mathsf{P}$*, then* $|\varphi| = 1$*; if* φ *is of the form* $\neg\psi$*, then* $|\varphi| = 1 + |\psi|$ *; and if* φ *is of the form* $(\psi \star \chi)$*,* $\star \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$*, then* $|\varphi| = 1 + |\psi| + |\chi|$*.*

The following definitions are needed later:

Definition 3. *A* position *is a word over the natural numbers, where* ε *denotes the empty word. If* $\pi = a_1 \ldots a_n$ *is a position and* $i \in \mathbb{N}$*, then* $i.\pi$ *denotes* $ia_1 \ldots a_n$ *and* $\pi.i$ *denotes* $a_1 \ldots a_n i$*. The* set of positions *of a formula* φ*,* $\mathsf{pos}(\varphi)$*, is defined inductively as follows:*

- *if* $\varphi \in \mathsf{P}$*, then* $\mathsf{pos}(\varphi) = \{\varepsilon\}$*;*
- *if* φ *is of the form* $\neg\varphi_1$*,* $\varphi_1 \wedge \ldots \wedge \varphi_n$*,* $\varphi_1 \vee \ldots \vee \varphi_n$*,* $\varphi_1 \Rightarrow \varphi_2$*, or* $\varphi_1 \Leftrightarrow \varphi_2$*, then* $\mathsf{pos}(\varphi) = \{\varepsilon\} \cup (\bigcup_{i=1}^{n} \{i.\pi \mid \pi \in \mathsf{pos}(\varphi_i)\})$*.*

Example 1. Let $\varphi = p \vee (q \wedge \neg r)$.

$$
\begin{aligned}
\mathsf{pos}(\neg r) &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \mathsf{pos}(r)\} = \{\varepsilon\} \cup \{1.\pi \mid \pi \in \{\varepsilon\}\}\{\varepsilon, 1\} \\
\mathsf{pos}(q \wedge \neg r) &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \mathsf{pos}(q)\} \cup \{2.\pi \mid \pi \in \mathsf{pos}(\neg r)\} \\
&= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \{\varepsilon\}\} \cup \{2.\pi \mid \pi \in \{\varepsilon, 1\}\} = \{\varepsilon, 1, 2, 2.1\} \\
\mathsf{pos}(\varphi) &= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \mathsf{pos}(p)\} \cup \{2.\pi \mid \pi \in \mathsf{pos}(q \wedge \neg r)\} \\
&= \{\varepsilon\} \cup \{1.\pi \mid \pi \in \{\varepsilon\}\} \cup \{2.\pi \mid \pi \in \{\varepsilon, 1, 2, 2.1\}\} \\
&= \{\varepsilon, 1, 2, 2.1, 2.2, 2.2.1\}
\end{aligned}
$$

We denote by $\varphi|_\pi$ the subformula of φ which starts at position π. Formally, if $\pi = \varepsilon$, then $\varphi|_\pi = \varphi$; if φ is of the form $\neg\varphi_1$, $\varphi_1 \wedge \ldots \wedge \varphi_n$, $\varphi_1 \vee \ldots \vee \varphi_n$, $\varphi_1 \Rightarrow \varphi_2$, or $\varphi_1 \Leftrightarrow \varphi_2$, and $\pi = i.\pi'$, for $i \in \mathbb{N}$ and position $\pi' \in \mathsf{pos}(\varphi_i)$, then $\varphi|_\pi = \varphi_i|_{\pi'}$. For instance, in Example 1, where $\varphi = p \vee (q \wedge \neg r)$, we have that $\varphi|_\varepsilon = \varphi$, $\varphi|_1 = p|_\varepsilon = p$, $\varphi|_2 = (q \wedge \neg r)|_\varepsilon = q \wedge \neg r$, $\varphi|_{2.1} = (q \wedge \neg r)|_1 = q|_\varepsilon = q$, and so on.

Definition 4. *Let* $\varphi \in \mathsf{WFF}$. *The* polarity *of a subformula of* φ*, starting at the position* π*,* $\mathsf{pol}(\varphi, \pi)$*, is given as follows:*

1. $\mathsf{pol}(\varphi, \varepsilon) = 1$.
2. *If* $\varphi|_\pi$ *is of the form* $\neg\varphi_1$*, then* $\mathsf{pol}(\varphi, \pi.1) = -\mathsf{pol}(\varphi, \pi)$.
3. *If* $\varphi|_\pi$ *is of the form* $\varphi_1 \wedge ... \wedge \varphi_n$*, or* $\varphi_1 \vee ... \vee \varphi_n$*, then* $\mathsf{pol}(\varphi, \pi.i) = \mathsf{pol}(\varphi, \pi)$*, for* $i = 1, ..., n$.
4. *If* $\varphi|_\pi$ *is of the form* $\varphi_1 \Rightarrow \varphi_2$*, then* $\mathsf{pol}(\varphi, \pi.1) = -\mathsf{pol}(\varphi, \pi)$ *and* $\mathsf{pol}(\varphi, \pi.2) = \mathsf{pol}(\varphi, \pi)$.
5. *If* $\varphi|_\pi$ *is of the form* $\varphi_1 \Leftrightarrow \varphi_2$*, then* $\mathsf{pol}(\varphi, \pi.1) = \mathsf{pol}(\varphi, \pi.2) = 0$.

Example 2. If $\varphi = p \Rightarrow (p \Leftrightarrow q)$, then $\mathsf{pol}(\varphi, \varepsilon) = 1$, $\mathsf{pol}(\varphi, 1) = -1$, $\mathsf{pol}(\varphi, 2) = 1$, $\mathsf{pol}(\varphi, 2.1) = 0$, and $\mathsf{pol}(\varphi, 2.2) = 0$.

A *valuation* is a function $\nu : \mathsf{WFF} \longrightarrow \{\mathit{true}, \mathit{false}\}$. A formula φ is said to be *satisfiable* if there is a valuation ν such that $\nu(\varphi) = \mathit{true}$. A formula φ is said to be *valid* if $\nu(\varphi) = \mathit{true}$ for any valuation ν. A formula is said to be *unsatisfiable* if there is no valuation ν such that $\nu(\varphi) = \mathit{true}$. Two formulae, φ and φ', are said to be semantically equivalent if, and only if, for any valuation ν we have that $\nu(\varphi) = \nu(\varphi')$. We denote by $\varphi \eqslantless\!\!\models \varphi'$ the fact that φ and φ' are semantically equivalent.

3 Normal Forms

A formula is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses. It is well known that for every formula φ there is a formula φ' such that φ' is in CNF and φ is semantically equivalent to φ'. The proof is by induction on the structure of a formula and uses the following semantic equivalences:

- $\varphi \Leftrightarrow \varphi' \eqslantless\!\!\models (\varphi \Rightarrow \varphi') \wedge (\varphi' \Rightarrow \varphi)$ (definition of double implication);
- $\varphi \Rightarrow \varphi' \eqslantless\!\!\models \neg\varphi \vee \varphi'$ (definition of implication);
- $\neg(\varphi \wedge \varphi') \eqslantless\!\!\models \neg\varphi \vee \neg\varphi'$ (De Morgan);
- $\neg(\varphi \vee \varphi') \eqslantless\!\!\models \neg\varphi \wedge \neg\varphi'$ (De Morgan);
- $\neg\neg\varphi \eqslantless\!\!\models \varphi$ (double negation elimination);
- $\varphi \vee (\varphi' \wedge \varphi'') \eqslantless\!\!\models (\varphi \vee \varphi') \wedge (\varphi \vee \varphi'')$ (distribution).

It is easy to see that any procedure for generating the CNF of a formula which is based on these equivalences might lead to a exponential blow up on the size of the original formula, because the result of applying the rewriting rule based on distribution adds twice the size of the formula which is being distributed. In order to avoid the undesirable growth in the size of the formula, *renaming* can be applied.

Definition 5. *Let* $\varphi \in \mathsf{WFF}$ *and* $\psi \sqsubset \varphi$*. A* renaming *of* φ *is a subset of proper subformulae of* φ*. Let* R *be a renaming of* φ *and let* P' *denote the set of propositional symbols occurring in* φ*. The substitution of* R *in* φ *is a function* $\sigma : R \longrightarrow \mathsf{P} \setminus \mathsf{P}'$*. If* $\sigma = \{(\varphi_1, p_1), \ldots, (\varphi_n, p_n)\}$ *is a substitution of* $R = \{\varphi_1, ..., \varphi_n\}$ *in* φ*, then:*

$$\mathsf{rep}(\varphi, \sigma) = \begin{cases} \varphi & \text{if } n = 0 \\ \mathsf{rep}(\mathsf{repb}(\varphi, \varphi_1, p_1), \{(\varphi_2, p_2), \ldots, (\varphi_n, p_n)\}) & \text{if } n > 0 \end{cases}$$

where $\mathsf{repb}(\varphi, \psi, p)$ *denotes the replacement of every occurrence of* ψ *in* φ *by* p*,* $p \notin P'$.

φ	$\mathsf{ncl}(\varphi)$	$\overline{\mathsf{ncl}}(\varphi)$
$\neg\varphi_1$	$\overline{\mathsf{ncl}}(\varphi_1)$	$\mathsf{ncl}(\varphi_1)$
$\varphi_1 \wedge ... \wedge \varphi_n$	$\sum_{i=1}^{n} \mathsf{ncl}(\varphi_i)$	$\prod_{i=1}^{n} \overline{\mathsf{ncl}}(\varphi_i)$
$\varphi_1 \vee ... \vee \varphi_n$	$\prod_{i=1}^{n} \mathsf{ncl}(\varphi_i)$	$\sum_{i=1}^{n} \overline{\mathsf{ncl}}(\varphi_i)$
$\varphi_1 \Rightarrow \varphi_2$	$\overline{\mathsf{ncl}}(\varphi_1)\mathsf{ncl}(\varphi_2)$	$\mathsf{ncl}(\varphi_1) + \overline{\mathsf{ncl}}(\varphi_2)$
$\varphi_1 \Leftrightarrow \varphi_2$	$\overline{\mathsf{ncl}}(\varphi_1)\mathsf{ncl}(\varphi_2) + \overline{\mathsf{ncl}}(\varphi_2)\mathsf{ncl}(\varphi_1)$	$\mathsf{ncl}(\varphi_1)\mathsf{ncl}(\varphi_2) + \overline{\mathsf{ncl}}(\varphi_1)\overline{\mathsf{ncl}}(\varphi_2)$
$\varphi \in \mathcal{P}$	1	1

Table 1: Number of clauses generated from a formula.

As functions can be seen as restrictions over binary relations, by convenience, we will often write σ as the set of pairs in $R \times \mathsf{P}$, as in the above definition.

Definition 6. *Let $\varphi \in \mathsf{WFF}$ and $\psi \sqsubset \varphi$. Let σ be a substitution of a renaming R of φ and $(\psi, p) \in \sigma$. The* definition of ψ in φ with respect to σ *is given as follows:*

$$\mathsf{def}(\varphi, \psi, \sigma) = \begin{cases} p \Rightarrow \xi & \textit{if } \mathsf{pol}(\varphi, \pi) = 1, \textit{ for all } \pi \in \mathsf{pos}(\varphi) \textit{ such that } \varphi|_\pi = \psi \\ \xi \Rightarrow p & \textit{if } \mathsf{pol}(\varphi, \pi) = -1, \textit{ for all } \pi \in \mathsf{pos}(\varphi) \textit{ such that } \varphi|_\pi = \psi \\ p \Leftrightarrow \xi & \textit{otherwise} \end{cases}$$

where $\xi = \mathsf{rep}(\psi, \sigma \setminus \{(\psi, p)\})$.

Note that $\mathsf{rep}(\psi, \sigma \setminus \{(\psi, p)\})$ denotes that all substitutions defined by $\sigma \setminus \{(\psi, p)\}$ have already been applied to ψ. In what follows, if a particular substitution $\sigma : R \longrightarrow P \setminus P'$ is not relevant in some context, we may simply write $\mathsf{def}(\varphi, \psi, R)$ for the definition of ψ in φ.

Let $\sigma = \{(\varphi_1, p_1), \ldots, (\varphi_n, p_n)\}$ be a substitution of $R = \{\varphi_1, \ldots, \varphi_n\}$ in φ, where R is a renaming of φ. The rewriting of φ as $\mathcal{R}(\varphi, \sigma)$, where $\mathcal{R}(\varphi, \sigma) = \mathsf{rep}(\varphi, \sigma) \wedge \mathsf{def}(\varphi, \varphi_1, \sigma) \wedge \ldots \wedge \mathsf{def}(\varphi, \varphi_n, \sigma)$, is called *transformation by renaming* and it is satisfiability preserving [17,13]. As the transformation adds a fixed number of symbols for each subformula of φ, the size of $\mathcal{R}(\varphi, \sigma)$ is linear in the size of φ. As an example, let $\varphi = (p \Leftrightarrow q) \Leftrightarrow (p \Leftrightarrow q)$ and $\sigma = \{(p \Leftrightarrow q, r)\}$. Then, $\mathcal{R}(\varphi, \sigma) = (r \Leftrightarrow r) \wedge (r \Leftrightarrow (p \Leftrightarrow q))$.

Let $\varphi \in \mathsf{WFF}$. Let $\mathsf{ncl}(\varphi)$ be the number of clauses obtained from the transformation into CNF by exhaustively applying the rewriting rules based on the semantic equivalences given in this section, that is, by applying the usual equivalence preserving algorithm to obtain the CNF of a formula. We denote by $\overline{\mathsf{ncl}}(\varphi)$ the number $\mathsf{ncl}(\neg\varphi)$. For any formula φ, $\mathsf{ncl}(\varphi)$ can be computed according to Table 1 (taken from [2]). Let $\sigma = \{(\varphi_1, p_1), \ldots, (\varphi_n, p_n)\}$ be a substitution of $R = \{\varphi_1, ..., \varphi_n\}$ in φ. We denote by $\mathsf{ncl}(\varphi, \sigma)$, or simply by $\mathsf{ncl}(\varphi, R)$, the number of clauses obtained for the transformation into CNF of $\mathcal{R}(\varphi, \sigma)$.

Denote by $\mathsf{SF}(\varphi)$ the multiset of subformulae of φ. Giving a renaming R in φ, the *benefit* of applying the renaming R in φ, denoted by $\mathsf{B}(R, \varphi)$ is given by $\mathsf{ncl}(\varphi) - \mathsf{ncl}(\mathcal{R}(\varphi, R))$, that is, the difference between the number of clauses generated by φ and

Form of ψ	$a^{\varphi}_{\psi_i}$	$b^{\varphi}_{\psi_i}$
$\neg\psi_1$	b^{φ}_{ψ}	a^{φ}_{ψ}
$\psi_1 \wedge ... \wedge \psi_n$	a^{φ}_{ψ}	$b^{\varphi}_{\psi} \prod_{j\neq i} \overline{\mathsf{ncl}}(\psi_j)$
$\psi_1 \vee ... \vee \psi_n$	$a^{\varphi}_{\psi} \prod_{j\neq i} \mathsf{ncl}(\psi_j)$	b^{φ}_{ψ}
$\psi_1 \Rightarrow \psi_2, i = 1$	b^{φ}_{ψ}	$a^{\varphi}_{\psi}\mathsf{ncl}(\psi_2)$
$\psi_1 \Rightarrow \psi_2, i = 2$	$a^{\varphi}_{\psi}\overline{\mathsf{ncl}}(\psi_1)$	b^{φ}_{ψ}
$\psi_1 \Leftrightarrow \psi_2, j = 3 - i$	$a^{\varphi}_{\psi}\overline{\mathsf{ncl}}(\psi_j) + b^{\varphi}_{\psi}\mathsf{ncl}(\psi_j)$	$a^{\varphi}_{\psi}\mathsf{ncl}(\psi_j) + b^{\varphi}_{\psi}\overline{\mathsf{ncl}}(\psi_j)$
$\psi_i = \varphi$	1	0

Table 2: Coefficients a and b.

the number of clauses generated by the result of the transformation by renaming of φ. We denote by $\mathsf{SUP}_{\mathsf{R}}(\varphi)$ the least element φ' in the sequence $\{\varphi' \in R \mid \varphi \sqsubseteq \varphi'\}$.

A renaming R is free of φ in ψ, with $\varphi \sqsubseteq \psi$, if, for all $\chi \in R$, $\mathsf{B}(\{\chi\}, \mathcal{R}(\psi, R \setminus \{\chi\})) \geq 0$. In other words, a renaming is free of a particular subformula if the benefit is positive even when this subformula is not considered for renaming. A renaming R, free of φ in ψ, is *top-down* if for all $\chi \sqsubseteq \varphi$, if $\mathsf{SUP}_(\chi)$ exists, then $\mathsf{B}(\{\chi\}, \mathsf{def}(\psi, \mathsf{SUP}_{\mathsf{R}}(\chi), R \setminus \mathsf{SF}(\chi))) < 0$, and else $\mathsf{B}(\{\chi\}, \mathcal{R}(\psi, R \setminus \mathsf{SF}(\chi))) < 0$. If $\varphi = \psi$, then R is top-down in ψ. This last conditions ensures that no subformula is renamed if it has a superformula of positive benefit which is not renamed.

4 Implementation and Evaluation

We note that the number of clauses generated from the transformation of a formula into CNF can be efficiently computed by using the coefficients given in Table 2. The coefficient a^{φ}_{ψ} (resp. b^{φ}_{ψ}) corresponds to the overall contribution of $\mathsf{ncl}(\varphi)$ in terms of $\mathsf{ncl}(\psi)$ (resp. $\overline{\mathsf{ncl}}(\psi)$). It can be shown [2] that in order to decide whether a subformula should be considered for renaming, we only need to check if $a^{\varphi}_{\psi} > 0$ and $b^{\varphi}_{\psi} > 0$.

Let $\{\varphi_1, ..., \varphi_n\}$ be the set of proper subformulae of φ and define $f(i, j)$, $i, j \leq n$, as a renaming $R \subseteq \{\varphi_1, \ldots, \varphi_i\}$ that contains at most j subformulae, that is $|R| \leq j$. By definition, we have that $f(i, 0) = f(0, j) = \emptyset$, for all i, j. Also, because we are interested in generating the renaming which will lead to the smallest number of clauses, we require that the inclusion of a formula in $f(i, j)$ is restricted as follows (for $i > 0$ and $j > 0$):

$$f(i,j) = \begin{cases} f(i-1, j-1) \cup \{\varphi_i\} & \text{if } \mathsf{ncl}(\varphi, f(i-1, j-1) \cup \{\varphi_i\}) < \mathsf{ncl}(\varphi, f(i-1, j)) \\ f(i-1, j) & \text{otherwise} \end{cases}$$

We can show that the definition of $f(i, j)$ does not enjoy an optimal substructure, that is, if we have that $f(i, j)$ is an optimal renaming among those which contain at most j subformulae and $\psi \in f(i, j)$, then $f(i, j) \setminus \{\psi\}$ might not be an optimal renaming with at most $j - 1$ subformulae. The counterexample follows. Take

$$\varphi = ((p_1 \wedge p_2 \wedge p_3 \wedge p_4) \vee (q_1 \wedge q_2)) \wedge ((r_1 \wedge r_2) \vee (s_1 \wedge ... \wedge s_{100})),$$

where $\mathsf{ncl}(\varphi) = 208$. Let $R = \{p_1 \wedge p_2 \wedge p_3 \wedge p_4, \psi = s_1 \wedge \ldots \wedge s_{100}\}$ is optimal and $|R| \leq 2$. However, $\mathsf{ncl}(\varphi, R - \{\psi\}) = 206 > 110 = \mathsf{ncl}(\phi, \{r_1 \wedge r_2\})$. That is, $R' = R - \{\psi\} = \{p_1 \wedge p_2 \wedge p_3 \wedge p_4\}$, $\psi \notin R'$ and $|R'| \leq 1$, but R' is not optimal. Thus, the renamings obtained through the computation of $f(n, n)$ might not be optimal. Still, the computation of $f(n, n)$ can be used as an heuristic in the search for good renamings.

Algorithm 1 computes $f(n, n)$ which is then stored in $dp[n]$. It is important to note that the computation of the renaming is done in a bottom-up fashion, but in our implementation, by construction, the candidates for a renaming in $\{\varphi_1, \ldots, \varphi_n\}$ have the property that if $\varphi_i \sqsubset \varphi_j$, then $i > j$, if the formula is linear. Although we are considering a carefully chosen ordering over the subformulae which are candidates for renaming, this does not ensure that the obtained renaming is a top-down one and, therefore, optimum (for linear formulae). Recall, that a top-down renaming requires that a formula being considered for renaming cannot have a superformula which is not considered for renaming. In particular, if in our implementation we use the representation of formulae as Directed Acyclic Graphs (DAGs) instead of trees, in a formula as $(p \wedge q \wedge r) \wedge ((p \wedge q \wedge r) \vee s)$, our procedure considers the renaming of $(p \wedge q \wedge r)$ even if $((p \wedge q \wedge r) \vee s)$ is not considered for renaming.

Algorithm 1 Bottom-up computation of $f(n, n)$.

```
1: Let dp[j] = ∅ for all j
2: for i = 1 to n do
3:     for j = n downto 1 do
4:         alt ← dp[j − 1] ∪ {φ_i}
5:         if ncl(φ, alt) < ncl(φ, dp[j]) then
6:             dp[j] ← alt
7:         end if
8:     end for
9: end for
```

The attentive reader has certainly noticed that the formula $(p \wedge q \wedge r) \wedge ((p \wedge q \wedge r) \vee s)$ is not linear. The example is still illustrative as it shows that we will not find a top-down renaming for every formula. However, in this case, the renaming given by the procedure in [2] generates more clauses than our procedure: their procedure generates six clauses $(p \wedge q \wedge r \wedge (s \vee p) \wedge (s \vee q) \wedge (s \vee r))$ whilst our procedure generates five $(t \wedge (t \vee s) \wedge (\neg t \vee p) \wedge (\neg t \vee q) \wedge (\neg t \vee r))$. In our experimental evaluation, we have found out that there are few examples in which their procedure generates less clauses than ours. In particular, when the formula has many occurrences of bi-implications, our procedure is more efficient.

We have implemented both algorithms and tested over a set of 1200 formulae [14]. The tests were performed on a PC with an Intel® Xeon®

Processor E5-2620 v3, 64 GiB RAM, 15 M Cache, running under GNU/Linux (3.19.0-30-generic x86_64). Virtual memory was limited to 4 GiB and the timeout was set to 1000s. Ten combinations of options were tested: either using DAGS or trees as data structures for representing formulae; either using simplification or not; either applying our algorithm or applying the algorithm in [2]. The other two options would allow to use the usual distribution rules for obtaining the CNF of a formula, with trees, using either our algorithm or the greedy one. The implementations and the raw data obtained from our implementations can be found at [4].

We briefly mention the most important results (full analysis can also be found at [4]). For the combination of options which uses DAGs without simplification, both algorithms produced outputs in 73% of the cases within the given time limit. In 3% of these cases, the algorithm proposed in [2] produced less clauses than our approach (the difference between the number of clauses being at most three). In 8% of those cases, our approach produced less clauses, but the difference between the number of clauses reached the peak of 1,572,786 clauses. The families of formulae where the differences were most notable were the families SYJ205 (where a formula is a conjunction of two semantically equivalent implications which are syntactically reordered), SYJ206 (where a formula is a bi-implication of two semantically equivalent bi-implications) and SYJ212 (which is similar to SYJ206, but double negations are applied to the first literal), where the number of occurrences of bi-implications is very high. Our results show that although our approach cannot be proved optimal, in practice it may perform better for unrestricted formulae.

5 Conclusions and Future Work

We have presented a procedure for transforming a formula into their CNF which combines both the usual rewriting rules and renaming. The idea behind the procedure is to avoid the renaming of subformulae whenever using this technique is not beneficial. Our implementation is based on dynamic programming, that is, we try to build a good renaming for a formula based on the renamings built for partial subproblems. As shown, our approach does not produce a top-down renaming, which would give us optimality for free in the case of linear formulae. In fact, our experimental evaluation has shown that the implementation of the greedy algorithm given in [2] performs better in some cases. However, when formulae are not restricted to linear ones, our approach outperforms the one in [2] for a larger subset of formulae and the difference between the number of clauses produced by our implementation is much smaller.

The proof that an optimal substructure for the problem exists is still ongoing work. We hope to prove that when formulae is restricted to linear formulae, we are able to construct a good ordering over the candidate subformulae for renaming which mimics the top-down approach in [2].

We also intend to extend our approach to deal with the transformation of modal formulae into the normal form given in [9] and add the procedure to the preprocessing options available in our existing prover [10]. In the case of modal formulae, the depth of a formula being considered for renaming also needs to be taken into consideration, making the decision of which technique to apply slightly more difficult.

Finally, we will investigate other properties related to the minimisation of normal forms. Although the number of clauses seems to be a good criteria, other measures could be taken into consideration as, for instance, measures related to the width of clauses (i.e. the number of literals in a clause).

References

1. R. Bloem, U. Egly, P. Klampfl, R. Könighofer, and F. Lonsing. SAT-based methods for circuit synthesis. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 31–34. FMCAD Inc, 2014.
2. T. Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4):283–301, 1992.
3. S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, 1971.
4. M. Costa de Sousa Carvalho Pimenta. Um algoritmo baseado em programação dinâmica e renomeamento para minimização de formas normais. Monografia de Conclusão de Curso, Bacharelado em Ciência da Computação, Universidade de Brasília, 2016. Available at `https://github.com/matheuscscp/TG`.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.
7. A. Gupta, M. K. Ganai, and C. Wang. SAT-based verification methods and applications in hardware verification. In *Formal Methods for Hardware Verification*, pages 108–143. Springer, 2006.
8. E. J. Horvitz. *Automated reasoning for biology and medicine*. Knowledge Systems Laboratory, Section on Medical Informatics, Stanford University, 1992.
9. C. Nalon, U. Hustadt, and C. Dixon. A modal-layered resolution calculus for K. In H. de Nivelle, editor, *Automated Reasoning with Analytic Tableaux and Related Methods - 24th International Conference, TABLEAUX 2015, Wrocław, Poland, September 21-24, 2015. Proceedings*, volume 9323 of *Lecture Notes in Computer Science*, pages 185–200. Springer, 2015.
10. C. Nalon, U. Hustadt, and C. Dixon. KSP: A resolution-based prover for multimodal K. In N. Olivetti and A. Tiwari, editors, *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings*, pages 406–415. Springer International Publishing, 2016.
11. R. Nieuwenhuis and A. Oliveras. On SAT modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 156–169. Springer, 2006.
12. A. Nonnengart and C. Weidenbach. Computing small clause normal forms. *Handbook of automated reasoning*, 1:335–367, 2001.
13. D. A. Plaisted and S. A. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Logic and Computation*, 2:293–304, 1986.
14. T. Raths, J. Otten, and C. Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1):261–271, 2007.
15. J. A. Robinson. Automatic Deduction with Hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.
16. J. A. Robinson. A Machine–Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, Jan. 1965.

17. G. Tseitin. On the Complexity of Derivations in Propositional Calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2*, Classical Papers on Computational Logic, pages 466–483. Springer-Verlag, 1983.

Comparação dos Métodos Scan Circular e Flexível na Detecção de Aglomerados Espaciais de Dengue

José C. S. Melo[1], Ana C. O. Melo[1], Ronei M. Moraes[1],

[1] Laboratório de Estatística Aplicada ao Processamento de Imagens e Geoprocessamento (LEAPIG),
Departamento de Estatística, Universidade Federal da Paraíba
João Pessoa, Paraíba, Brasil
zka07@hotmail.com, anaclaudiaemelo@gmail.com, ronei@de.ufpb.br

Abstract. A detecção de aglomerados espaciais é útil para identificar localidades com valores diferenciados significativos ou não do ponto de vista estatístico em uma região geográfica de interesse. Do ponto de vista epidemiológico, essa detecção auxilia a promover políticas públicas diferenciadas para o combate à uma doença. Neste artigo objetivou-se comparar o desempenho das Estatísticas Scan Circular e Scan Flexível para detecção de aglomerados espaciais usando dados reais de dengue na Paraíba.

Keywords: Métodos de Aglomeração Espacial, Scan Circular, Estatística Scan Flexível, Epidemiologia do Dengue.

1 Introdução

A Epidemiologia é a ciência que estuda uma doença segundo os seus padrões, causas e efeitos. Ela visa prover a base de conhecimento para a promoção e cuidados em saúde de acordo com as especificidades de cada localidade e de sua população específica [1]. Ela pode ainda auxiliar a tomada de decisão em saúde por gestores de modo a prover diferentes políticas de acordo com os dados epidemiológicos de cada localidade [2], elegendo diferentes níveis de prioridade para cada localidade de acordo com a região geográfica na qual ela está inserida [3]. Problemas como esse, remetem ao uso dos métodos de aglomeração espacial, que se utilizam de informações georreferenciadas para identificar localidades com valores diferenciados significativos ou não do ponto de vista estatístico.

Vários métodos para detecção de aglomerados espaciais estão disponíveis na literatura científica. Alguns são baseados em matrizes de proximidade, como o Índice de Moran e a Estatística de Getis & Ord [4]. Outros são baseados em grafos de vizinhança, como a Estatística Scan Circular [5] e a Estatística Scan Flexível [6]. Em situações práticas, os métodos são baseados em metodologias diferentes e, portanto, produzem resultados diferentes. Além disso, não há uma informação de referência para se avaliar quais aglomerados são verdadeiros ou não. Assim, são usadas formas indiretas de avaliação, baseadas por exemplo nos mapas de risco [7].

O mosquito Aedes aegypti, que é o vetor transmissor do dengue, foi detectado nas principais cidades do Brasil na década de 1970, depois de ter sido erradicado na década de 1950 [8]. O combate à doença é mais efetivo se for possível detectar a presença do vetor. Do ponto de vista epidemiológico, isso se concretiza a partir da detecção de aglomerados espaciais da doença na região geográfica em estudo e promovendo políticas públicas direcionadas àquelas sub-regiões.

Portanto, dado que essa detecção é fundamental para direcionar políticas que possam ser efetivas no combate à doença, torna-se necessário usar a melhor metodologia disponível para tal. Tango e Takahashi afirmam que a Estatística Scan Flexível, proposta por eles, classifica os conglomerados não circulares de maneira mais eficiente que a Estatística Scan Circular proposta por Kulldorff [5]. Desse modo, esse artigo visa comparar o desempenho dos métodos Scan Circular e Scan Flexível na detecção de aglomerados espaciais de dengue na Paraíba, usando para isso dados notificados no ano de 2013

2 Métodos

2.1 A Estatística Scan Circular

Considerando a situação em que, a região geográfica em estudo é dividida em m sub-regiões ou geo-objetos (por exemplo, municípios, distritos, bairros, etc). O número de casos na sub-região i é denotado pela variável aleatória N_i com valor observado n_i $(i = 1, \dots, m)$ e $\mathrm{n} = \mathrm{n}_1 + \cdots + \mathrm{n}_\mathrm{m}$. A hipótese H_0 afirma que não há aglomerados espaciais na sub-região i, os N_i são variáveis independentes de Poisson tal que

$$\mathrm{H}_0 : \mathrm{E}(\mathrm{N}_\mathrm{i}) = \xi_\mathrm{i}, \mathrm{N}_\mathrm{i} \sim \mathrm{Poisson}(\xi_\mathrm{i}), \mathrm{i} = 1, \dots, \mathrm{m}, \quad (1)$$

onde $Poisson(\xi)$ denota uma distribuição de Poisson com média ξ, e o ξ_i é o número esperado de casos da sub-região i sob a hipótese nula. Assim calculamos ξ_i como

$$\xi_\mathrm{i} = \mathrm{n} \frac{\mathrm{w_i}}{\sum_{\mathrm{k}=1}^{\mathrm{m}} \mathrm{w_k}}, \ \mathrm{i} = 1, \dots, \mathrm{m}, \quad (2)$$

onde w_i denota o tamanho da população na sub-região i. Usaremos as coordenadas do centro do município para especificar a posição geográfica de cada sub-região i [9].

Kulldorff [5] propõe, para a situação descrita acima, a estatística scan circular que gera uma janela Z em cada centroide das sub-regiões, que é o ponto centro geométrico de uma sub-região. Neste caso, uma janela consiste no círculo criado a partir do centroide. Para qualquer destes centroides, o raio do círculo varia continuamente desde zero até um percentual da população em risco a ser coberta, estabelecido pelo usuário. Logo se uma janela contém o centroide de uma sub-região, o raio do círculo crescerá até englobar, nesta janela, o percentual da população estabelecido. Seja $\mathrm{Z_{ik}}(\mathrm{k} = 1, \dots, \mathrm{K_i})$ denotando a janela composta pelos

k-1 vizinhos à sub-região *i,* então todas as janelas a serem verificadas pela estatística scan circular estão incluídas no conjunto

$$Z = Z_1 = \{Z_{ik} | 1 \leq i \leq m, 1 \leq k \leq K_i\}.$$

Com a utilização da notação da janela $Z \in Z$, a hipótese nula (1) é expressa como

$$H_0 : E\big(N(Z)\big) = \xi(Z), \text{ para todo } Z \in Z, \quad (3)$$

onde $N()$ e $\xi()$ denotam, respectivamente, a variável aleatória para o número de casos e o número esperado de casos sob H_0 dentro da janela especificada. A hipótese alternativa H_1, afirma que existe pelo menos uma janela $Z \in Z$, para o qual o risco é mais elevado no interior da janela, quando comparado com o exterior da mesma, que é,

$$H_1 : E\big(N(Z)\big) > \xi(Z), \text{ para algum } Z \in Z, \quad (4)$$

É possível calcular a probabilidade de observar o número de casos observados dentro e fora da janela, respectivamente para cada janela de Z. Kulldorff [5] propõe que sob H_0, a estatística da razão de verossimilhança é calculada por

$$\lambda_K = \max_{Z \in Z} \lambda_K(Z) = \max_{Z \in Z} \left(\frac{n(Z)}{\xi(Z)}\right)^{n(Z)} \left(\frac{n-n(Z)}{n-\xi(Z)}\right)^{n-n(Z)} I\left(\frac{n(Z)}{\xi(Z)} > \frac{n-n(Z)}{n-\xi(Z)}\right), \quad (5)$$

onde *n()* denota o número de casos observados dentro da janela especificada e *I()* é a função indicadora.

2.2 A Estatística Scan Flexível

A proposta de Tango e Takahashi [6] é de criar uma janela de forma flexível em cada centroide da sub-região, ligando as sub-regiões vizinhas. O processo, portanto ocorre da seguinte forma, para qualquer sub-região *i*, criamos o conjunto de janelas de forma flexível com comprimento *k*, o que consiste em *k* sub-regiões conectadas incluindo *i* e vamos mover *k* de 1 até o comprimento máximo pré-estabelecido K de vizinhos mais próximos. Para evitar a detecção de um conjunto de forma improvável, as sub-regiões ligadas são restritas aos subconjuntos do conjunto de sub-regiões *i* e os K vizinhos mais próximos à região *i.* Ao final, como na estatística scan circular, várias janelas diferentes de formas arbitraria e sobrepostas umas às outras, são criadas. Seja $Z_{ik(j)}$, $j = 1, \dots, j_{ik}$ denotando a janela de ordem *j*, a qual é um conjunto de *k* sub-regiões conectadas a partir da sub-região *i*, onde j_{ik} é a janela *j* satisfazendo $Z_{iK(j)} \subseteq Z_{iK}$ para $k = 1, \dots, K_i = K$. Então, todas as janelas a serem verificados são incluídas no conjunto

$$\mathcal{Z} = \mathcal{Z}_2 = \{\mathbb{Z}_{ik(j)} | 1 \leq i \leq m, 1 \leq k \leq K_i, 1 \leq j \leq j_{ik}\}.$$

De forma mais clara, para qualquer sub-região *i*, a estatística scan circular considera *K* círculos concêntricos que denotamos por Z_1, enquanto que a estatística scan flexível considera *K* círculos concêntricos mais todos os conjuntos de sub-regiões ligados (incluindo a única região *i*) cujos centroides estão localizados dentro do K-ésimo maior círculo concêntrico que denotamos por .

Portanto, o tamanho de $\mathcal{Z}_2$ é muito maior do que o de, que é no máximo mK. Outro ponto a ser destacado é que, o comprimento máximo de K deve ser inferior a 30, pois a carga computacional, devido ao grande número de possíveis combinações de janelas tornar-se-ia muito pesada. O valor de K, padrão no *software* FleXScan [6] é definido como 15.

A janela Z^* que contem a razão de máxima verossimilhança é definida como a *MLC,* ou seja, o aglomerado mais provável. No entanto, não é interessante que Z^*continue aumentando o seu raio, quando já englobou em seu círculo os aglomerados espaciais de maior risco, apenas para atingir o percentual da população pré-estabelecido pelo usuário, pois desta forma, englobará na mesma janela também aglomerados espaciais de menor risco [6, 10]. Tango [11] propôs que no processo de varredura na janela baseada em $\lambda_K(Z)$, exista uma possibilidade de que existam duas janelas disjuntas $\mathbb{Z}_1$ e Z_2 e várias regiões $\{i_1\}, \dots, \{i_r\}$ tal que

$$\lambda_k\left(\{\mathbb{Z}_1, \mathbb{Z}_2, \{i_1\}, \dots, \{i_r\}\}\right) > \max\{\lambda_k(\mathbb{Z}_1), \lambda_k(\mathbb{Z}_2)\}, \quad (6)$$

onde

$$\frac{n(\mathbb{Z}_1)}{\xi(\mathbb{Z}_1)} > 1,\ \frac{n(\mathbb{Z}_2)}{\xi(\mathbb{Z}_2)} > 1 \quad e \quad \frac{n_i}{\xi_i} \leq 1\ (i = 1, \dots, r).$$

Para evitar fenômenos indesejáveis, Tango [5] propôs a seguinte razão de verossimilhança restrita considerando, para cada sub-região, um risco individual:

$$\lambda_T(Z) = \left(\frac{n(Z)}{\xi(Z)}\right)^{n(Z)} \left(\frac{n-n(Z)}{n-\xi(Z)}\right)^{n-n(Z)} I\left(\frac{n(Z)}{\xi(Z)} > \frac{n-n(Z)}{n-\xi(Z)}\right) \prod_{i \in Z} I(p_i < \alpha_1), \quad (7)$$

onde p_i é o p-*value* uni caudal do teste para $H_0 : E(N_i) = \xi_i$ e é dado pelo p-*value* médio.

$$p_i = P_r\{N_i \geq n_i + 1 | N_i \sim \text{Poisson}(\xi_i)\} + \frac{1}{2} P_r\{N_i = n_i | N_i \sim \text{Poisson}(\xi_i)\}, \quad (8)$$

onde a função indicadora $I(p_i < \alpha_1)$ funciona como critério de seleção para os valores que estão na fronteira e α_1 é o nível de significância pré-definido para a região individual. A opção de usar o *p-value* médio é para ajustar o *p-valor* para pequenos ξ_i e contar os resultados. Por conseguinte, tal como no caso da estatística scan flexível original, o valor de p do teste scan flexível com base na razão de verossimilhança (7) é obtido através do teste da hipótese de Monte Carlo.

Para analisar os mapas das estatísticas espaciais, toma-se como referência o mapa de Risco Relativo (RR). Para a obtenção do mapa coroplético (como é denominado na Geografia qualquer mapa colorido) que seja referente ao risco do dengue no estado, se faz necessário o cálculo do RR. Este, por sua vez, permitirá a comparação da informação de diferentes áreas, padronizando os dados e, com isso, retirando o efeito das diferentes populações. Este indicador representa a intensidade da ocorrência de um fenômeno com relação a todas as regiões de estudo [3]. A equação do RR de uma área é denotada por:

$$RR_i = \frac{x_i/n_i}{\sum x_i / \sum n_i}, (i = 1, \dots, m) \quad (9)$$

onde x_i é o número de ocorrência do fenômeno em uma região e n_i é a população dessa região.

3 Resultados

Observando o mapa de RR (Fig. 1), nota-se que nas regiões leste e centro-sul possuem risco relativo baixo (inferior a 0,5 vezes o risco do Estado da Paraíba) em comparação com as demais áreas do mapa. Dos 223 municípios da Paraíba, 52 possuem risco elevado (superior ou igual a 1,5 vezes o risco do Estado da Paraíba) e cerca de 20 municípios não apresentaram risco. Quando se tem risco igual ou próximo a 1 significa que o risco do município é o mesmo que o do Estado. Por conseguinte, risco igual a 0 indica que não foi observado risco no município levando em conta o risco do Estado. Observa-se no mapa de RR que o maior número de municípios que possuem risco alto encontra-se ao oeste, havendo uma pequena concentração na parte central do Estado.

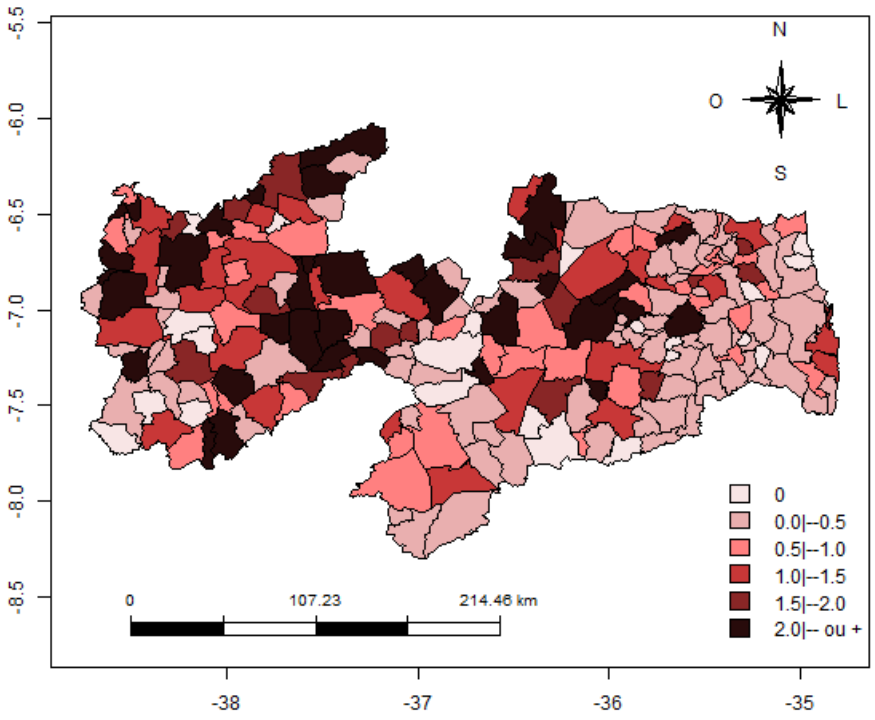


Fig. 1. Mapa do Risco Relativo para o dengue no ano de 2013 na Paraíba.

Analisando o método Scan Flexível (Fig. 2), observa-se que para cada aglomerado significativo que é detectado atribui-se uma cor, essa cor diferencia o aglomerado de forma que possam ser identificados os municípios que se relacionam. Com respeito à epidemiologia do dengue, nota-se a disparidade entre a área litorânea do estado (ao leste) e a área que representa o sertão paraibano (ao oeste), no que diz respeito ao número de municípios detectados. No litoral foram detectados poucos municípios, enquanto a maioria foi detectada no sertão.

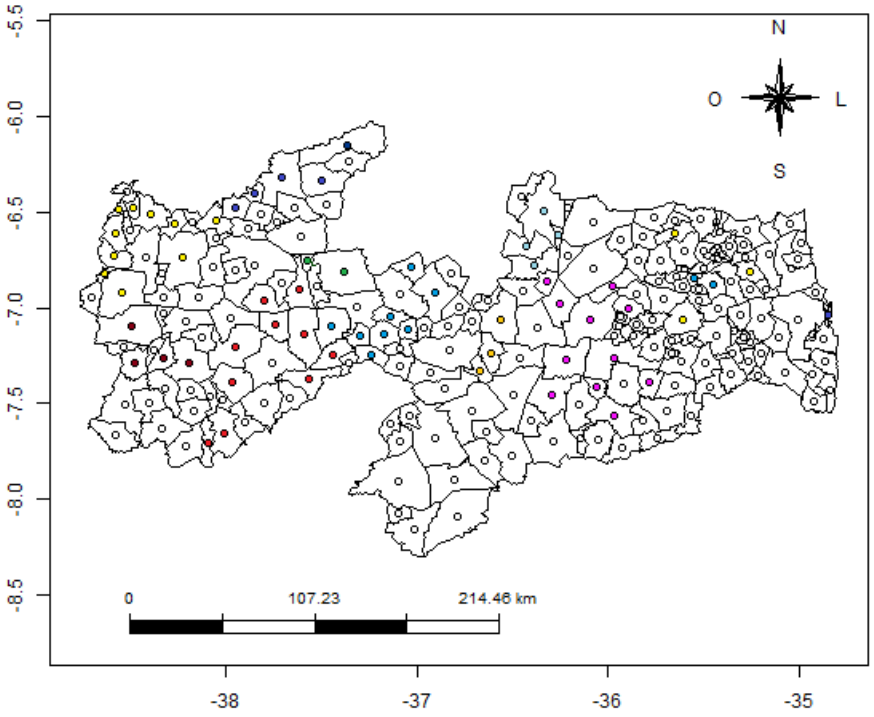


Fig. 2. Resultado da Estatística Scan Flexível para o dengue no ano de 2013 na Paraíba.

A Estatística Scan Circular (Fig. 3) apresentou resultado similar a Flexível, no qual a parte oeste do estado apresentou o maior número de municípios detectados. Todavia, o Scan detectou nove municípios a mais do que o método flexível. Também vale ressaltar que a parte centro-sul do estado, em ambos os métodos, não foram detectados municípios, o que é compatível com o mapa de RR (Fig. 1).

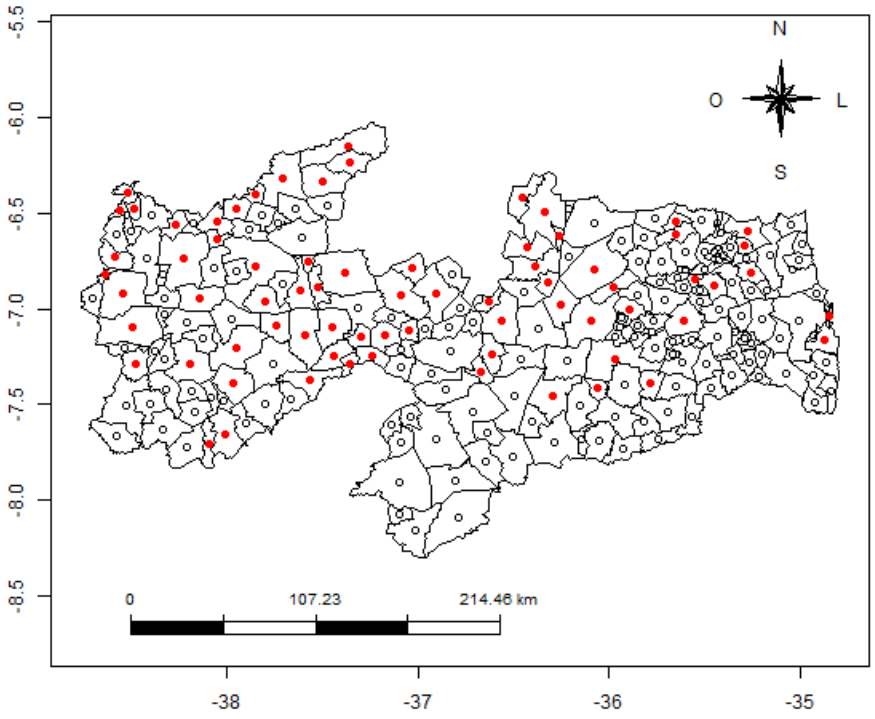


Fig. 3. Resultado da Estatística Scan Circular para o dengue no ano de 2013 na Paraíba.

Levando-se em conta o mapa RR como referência, a Estatística Scan Flexível detectou 63 municípios, sendo 50 deles valores de risco alto, deixando de detectar 2 municípios nessa situação. Ele também incluiu 3 municípios de risco baixo e 10 risco aproximadamente igual ao risco do Estado nesses aglomerados. A Estatística Scan Circular detectou 72 municípios, dos quais 52 deles apresentam risco alto. Observa-se que a Estatística Scan Circular conseguiu detectar todas as áreas de alto risco, quando comparado ao mapa de RR, porém também incluiu 3 municípios de risco baixo (onde apenas um deles foi também detectado pela Estatística Scan Flexível) e 17 risco aproximadamente igual ao risco do Estado (sendo 5 deles também detectados pela Estatística Scan Flexível) nesses aglomerados.

4 Conclusão

Levando em consideração o mapa RR como referência, ambos os métodos superestimaram os número de municípios com valores de risco alto. A Estatística Scan Flexível, entretanto, não detectou dois desses municípios de valores de risco alto,

enquanto a Estatística Scan Circular o fez para todos eles. A Estatística Scan Flexível adicionou aos seus aglomerados 13 municípios com valores de risco baixo ou risco aproximadamente igual ao risco do Estado. Em contrapartida, a Estatística Scan Circular incluiu 20 municípios cujos valores de risco não são altos nos seus aglomerados. Portanto, para a epidemiologia do dengue analisada neste trabalho, ambas as formas de Estatística Scan superestimaram os aglomerados espaciais detectados, mas a Estatística Scan Flexível o fez para um número menor de casos. Essas diferenças certamente se devem ao formato geométrico da janela utilizada por cada método.

Como trabalhos futuros, espera-se estender essa comparação para a epidemiologia de outras doenças de modo a aprofundar o conhecimento das vantagens e desvantagens de cada método em várias situações distintas.

Referências

1. Bailey, L.; Vardulaki, K.; Langham, J.; Chandramohan, D. Introduction to Epidemiology, 1st ed. London: Open University Press (2007).

2. Sanderson C.; Gruen, R. Analytical Models for Decision Making. London: Open University Press (2006).

3. Rothman, K.; Lash, T.; Greenland, S. Modern Epidemiology. Wolters Kluwer (2012).

4. Anselin, L. Spatial data analysis with GIS: an introduction to application in the social sciences. National Center for Geographic Information end Anlisis. University of California - Santa Barbara. August (1992).

5. Kulldorff M. A spatial scan statistic. Communications in Statistics: Theory and Methods; 26:1481--1496 (1997)

6. Tango T, Takahashi K. A Flexibly Shaped Spatial Scan Statistic for Detecting Clusters. International Journal of Health Geographics; 4:11. DOI: 10.1186/1476-072X-4-11 (2005)

7. Moraes, R. M.; Nogueira, J. A. & Sousa, A. C. A. A New Architecture for a Spatio-Temporal Decision Support System for Epidemiological Purposes, in Proceedings of the 11th International FLINS Conference on Decision Making and Soft Computing (FLINS2014), Agosto, , Brazil, pp. 17--23 (2014)

8. Braga, I. A.; Valle, D. Aedes Aegypti: Histórico do Controle no Brasil. Epidemiologia e Serviços de Saúde, 16(2), 113--118 (2007)

9. Tango, T., & Takahashi, K.. A Flexible Spatial Scan Statistic with a Restricted Likelihood Ratio for Detecting Disease Clusters. Statistics in medicine, 31(30), 4207--4218 (2012)
10. Tango T. A Test for Spatial Disease Clustering Adjusted for Multiple Testing. Statistics in Medicine; 19:191--204. DOI: 10.1002/(SICI)1097-0258(20000130) (2000)
11. Tango T. A Spatial Scan Statistic With a Restricted Likelihood Ratio. Japanese Journal of Biometrics; 29:75--95 (2008)

Notes on Topoi and Refinement

Christiano Braga[1] and E. Hermann Haeusler[2]

[1] Universidade Federal Fluminense
cbraga@ic.uff.br
[2] Pontifícia Universidade Católica do Rio de Janeiro
hermann@inf.puc-rio.br

Abstract. Category Theory is an appropriate framework to establish relations among concepts that are perhaps in very different spheres of the human knowledge, such as a biological or physical concept and a computation model. It is not surprising then that it has been thoroughly applied to relate concepts in Computer Science, in particular specifications or programs. Topoi is a theory that considers a particular kind of category. In one of its axiomatic definitions, a topos has a terminal object, all pullbacks, an exponential and a subobject classifier. Moreover, a topos has an internal logic that can be used to express properties local to a given topos, called Local Set Theory. We propose to explore the internal logic of a topos to specify and prove refinements among software components.

1 Introduction

We propose a study of software refinement using Topoi [9] and the Local Set Theory, the logic internal to each topos. (The use of LST in software refinement appears not to be very exercised in the literature.) We use a monoid actions or M-set to represent an automaton (which we consider to be the semantics of a software component) and the fact that the category of M-sets, denoted **M-Set**, is a topos. We interpret refinement as the subautomaton relation which in **M-Set** is interpreted as an injection morphism. LST allows for the specification of different properties about automata and their relations.

We believe that this manuscript fits nicely with ETMF because, at the same time, it reports on an ongoing work, a research on refinement, and presents Topoi concepts in a tutorialistic fashion. The study of Category Theory and Topoi is usually quite dry, based solely on mathematical examples for each categorical construction. To study them from automata theory perspective appears to be an appealing approach to Computer Science. The tutorial part of this paper, in Section 4, is based on [9, 10, 18].

The remainder of this short paper is organized as follows. We continue in Section 2 with some related work. Section 4 recalls basic definitions and results about Category Theory, Topoi and the **M-Set** category. Section 5 discusses initial thoughts on how to represent refinement in **M-Set** and how the Lean Theorem Prover could be used in the specification and proof of refinements in **M-Set**. We conclude this short paper in Section 6 with our final remarks and possible continuations of this work.

2 Some Related Work

To the best of our knowledge, Topoi, as opposed to general Category Theory, is not much explored in the context of specification refinement. Kestrel's Specware [20] approach is perhaps the only one where refinement is interpreted as a sheaf. However, they do not explore the internal logic of a topos to formalize refinement.

The literature is quite rich on refinement in general. Here we recall some of it together with hybrid and behavioral specifications that we believe to be closely connected subjects.

In a recent paper [5], Castro and Aguirre explore 2-categories and institutions to formalize refinement of specifications. In [12], Naumann and others explore data refinement and lax natural transformations, in the context of semantics of programming languages. In [14], Barbosa and others discuss about refinement in hybridised institutions. In [17], Orejas and others discuss early work on refinement and the problem of composition of implementations.

The B method [1] is a stablished approach to component refinement. The semantics of B machines is a set-theoretic one. The Unifying Theory of Programming [11] is also a representative approach to software refinement which is formalized as universal inverse implication. In [6] Cavalcanti presents in her PhD thesis a refinement calculus for the Z specification language. Sampaio and others described in [19] refinement in Circus, a language that integrates CSP, Z and the refinement calculus. In a recent paper, Sampaio and others study refinement of SysML models [13].

On heterogeneous specifications, the Hets [16] approach of Mossakowsky, based on Diaconescu's Grothendik institutions [7], is a quite relevant one. In [15] Jose Meseguer discusses "what is a logic?" also in the context of heterogeneous specifications. Finally, we refer to Goguen's behavioral specifications based on Hidden Algebra [8].

3 Subautomaton, Refinement and M-sets

As mentioned in the Introduction, we consider in this paper that a software component has an automaton semantics. Refinement between components is thus defined model-theoretically by the subautomaton relation between deterministic finite automata (DFA) that model each component. This is formalized by Definitions 1 and 2.

Definition 1 (Subautomaton). *An automaton* $H = (Q_h, \Sigma, \delta_h, q_{0_h}, Q_{f_h})$ *is a subautomaton of* $G = (Q_g, \Sigma, \delta_g, q_{0_g}, Q_{f_g})$ *denoted* $H \sqsubseteq G$ *iff*

$$Q_h \subseteq Q_g, q_{0_h} = q_{0_g}, Q_{f_h} = (Q_{f_g} \cap Q_h) \text{ and } p \in \delta_h(q, \sigma) \Rightarrow p \in \delta_g(q, \sigma).$$

Definition 2 (Refinement). *Given two components* A *and* C*,*

$$C \textbf{ refines } A \equiv M_A \sqsubseteq M_C$$

where M_i *is the deterministic finite automaton that gives semantics to component* i*.*

Coffee brewing	
Press on/off button ⏻.	The associated led blinks while water heats. The coffee maker is ready for use when the led stops blinking and remains lit.
Press the button that corresponds to the number of cups you want.	Press ☕ for a cup and ☕☕ for two cups or a mug. The coffee maker starts to brew the coffee, The coffee maker uses a standard amount of water for each cup. You may interrupt the brewing process at any moment by pressing the on/off button ⏻. Once you restart the coffee maker after interrupting the brewing process, it will not conclude the interrupted cycle.
Problem	
The associated led blinks quickly.	Make sure that there is enough water in the water compartment.

Table 1: Coffee maker behavior

Example 1 (Refinement). The automaton in Figure 1b represents (part of) the behavior of a real coffee machine whose natural language description is given in Table 1. Its alphabet is $\Sigma = \{⏻, ☕, ☕☕, warmed, brewed, emptied\}$. They denote actions that either a user may apply to the coffee maker (such as ⏻or ☕to switch the machine on or off, or to request a cup of coffee, respectively) or "internal" ones (such as *warmed*, denoting it may start making coffee, or *emptied*, denoting that the water reservoir is empty). The set of states is

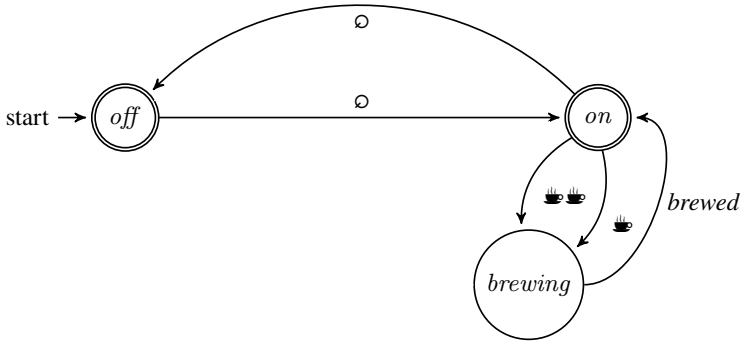
$$Q = \{on, off, warming, empty, brewing\}.$$

The transition function is graphically represented by the directed graph with distinguished notes in Figure 1b.

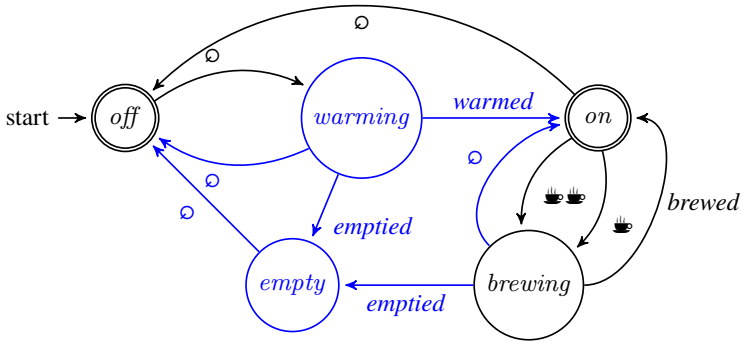
Let us consider now the automaton in Figure 1a. It simply specifies that a coffee maker may be switched on, prepare some coffee and be switched off. It is easy to see that the automaton of Figure 1a is a subautomaton of the one in Figure 1b. Let us call them A and C, respectively, for abstract and concrete. It is the case that $A \sqsubseteq C$. The set of states of A is a subset of the states of C, they both have the same initial state, the final states of $Q_{f_A} = Q_{f_C} \cap Q_A$, finally, for the every transition in A for given state and symbol, there exists a transition in C for the same source and symbol. Note that transition $off \xrightarrow{⏻}_A on$, in the abstract automaton in Figure 1a, is refined by $off\,(\xrightarrow{⏻}_C \circ\ warmed)\ on$, in the concrete automaton in Figure 1b.

Definition 3 (M-*set*). *Let $M = (A, *, e)$ be a monoid. An M-set is defined to be a pair (X, λ) where X is a set, $\lambda : A \times X \to X$ is an action of M on X, $m, p \in A$ and λ defined as*

$$\lambda_e(x) = x,$$
$$\lambda_m(\lambda_p(x)) = \lambda_{m*p}(x).$$



(a) Abstract automaton for a coffee maker



(b) Refined automaton for a coffee maker

Fig. 1: Automata for a coffee maker

The transition function of C may be represented by Equations 1 to 10 when we understand the set $X = Q$ and $A = \Sigma$, with a similar set for DFA A.

$$\lambda_{\text{☺}}(off) = warming \tag{1}$$
$$\lambda_{warmed}(warming) = on \tag{2}$$
$$\lambda_{emptied}(warming) = empty \tag{3}$$
$$\lambda_{\text{☺}}(on) = off \tag{4}$$
$$\lambda_{\text{☕}}(on) = brewing \tag{5}$$
$$\lambda_{\text{☕☕}}(on) = brewing \tag{6}$$
$$\lambda_{\text{☺}}(warming) = off \tag{7}$$
$$\lambda_{brewed}(brewing) = on \tag{8}$$
$$\lambda_{emptied}(brewing) = empty \tag{9}$$
$$\lambda_{\text{☺}}(empty) = off \tag{10}$$

In the next section we will see that M-sets and M-sets homomorphisms give rise to a category that is also a topos. We may then specify properties about automata, such as in

reachability analysis, simulation or refinement, in the internal logic of the topos, called Local Set Theory.

4 Category Theory and Topoi

4.1 Category theory

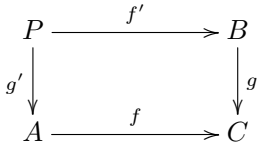
Definition 4 (Category). *A category* ***C*** *comprises*

- *a collection of things called* ***C****-objects;*
- *a collection of things called* ***C****-arrows;*
- *operations asigning to each* ***C****-arrow* f *a* ***C****–object* $domf$ *(the domain of* f*) and a* ***C****–object* $codf$ *(the codomain of* f*). If* $a = domf$ *and* $b = codf$ *it is displayed as* $f : a \to b$ *or* $a \xrightarrow{f} b$*;*
- *an operation assigning to each pair* $\langle g, f\rangle$ *of* ***C****-arrows with* $domg = codf$*, a* ***C****–arrow* $g \circ f$*, the composite of* f *and* g*, having* $domg \circ f = domf$ *and* $codg \circ f = codg$*, such that given the configuration* $a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{h} d$ *of* $\mathbf{C}$*-objects and* $\mathbf{C}$*-arrows then* $h \circ (g \circ f) = (h \circ g) \circ f$*;*
- *an assignment to each* $\mathbf{C}$*-object* b *to a* $\mathbf{C}$*-arrow* $1_b : b \to b$ *called the identity arrow on* b *such that for any* $\mathbf{C}$*-arrows* $f : a \to b$ *and* $g : b \to c$ $1_b \circ f = f, g \circ 1_b = g$*.*

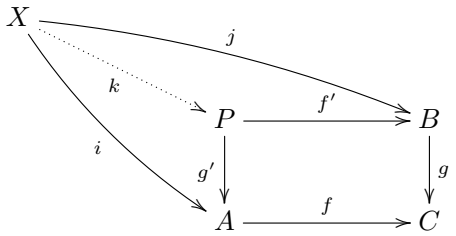
Definition 5 (Terminal). *An object* $\mathbf{1}$ *is a* terminal *object if for every object* A *in a category there exists a single morphism between* A *and* $\mathbf{1}$*, denoted* $A \xrightarrow{!} \mathbf{1}$*.*

Example 2 (Terminal). In the category of sets $\mathbf{Set}$, terminal objects form a categorical counterpart of the elements of a set S. Functions from a singleton set to S are in a 1-to-1 correspondence with the elements of S.

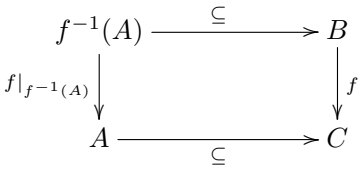
Definition 6 (Pullback). *A* pullback *or fiber product of the pair of arrows* $f : A \to C$ *and* $g : B \to C$ *is an object* P *and a pair of arrows* $g' : P \to A$ *and* $f' : P \to B$ *such that* $f \circ g' = g \circ f'$



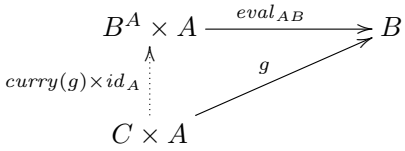
and if $i : X \to A$ *and* $j : X \to B$ *such that* $f \circ i = g \circ j$ *then there is a unique* $k : X \to P$ *such that* $i = g' \circ k$ *and* $j = f' \circ k$*:*



Example 3 (Pullback). Let $f : B \to C$ be a function in $\mathbf{Set}$ and $A \subseteq C$. The inverse image of A under f is written $f^{-1}(A) = \{b \mid f(b) \in A\}$. We denote $f|_S$ for the restriction of f to a set $S \subseteq B$.



Definition 7 (Exponential). *Let* $\mathbf{C}$ *be a category with all binary products and let* A *and* B *be objects of* $\mathbf{C}$*. An object* B^A *is a* exponential *object if there is an arrow* $eval_{AB} : (B^A \times A \to B)$ *such that for any object* C *and arrow* $g : (C \times A) \to B$ *there is a unique arrow* $curry(g) : C \to B^A$ *making the following diagram commute,*



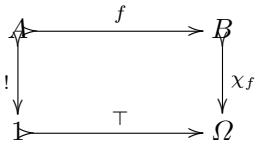
that is, a unique arrow $curry(g)$ *such that*

$$eval_{AB} \circ (curry(g) \times id_A) = g.$$

Note 1 (Exponential). The exponential construction gives a categorical interpretation to the notion of *currying*.

Example 4 (Exponential). A cartesian closed category is a category with a terminal object, all binary products and exponentiation. The category $\mathbf{Set}$ is cartesian closed with $B^A = \mathbf{Set}(A, B)$.

Definition 8 (Subobject classifier). *A sub-object classifier, in a category* $\mathbf{C}$*, is an object* Ω*, together with a morphism* $\top : 1 \to \Omega$*, such that, for every monomorphism* $f : B \to A$*, there is a unique morphism* $\chi_f : A \to \Omega$*, such that, the following diagram is a pullback.*



Example 5 (Subobject classifier). The characteristic function of a set.

Example 6 (Subobject classifier of $\mathbf{M}$-$\mathbf{Set}$*).* To illustrate the workings of the subobject classifier, suppose $k : (X, \lambda) \to (Y, \mu)$ to be the inclusion $X \hookrightarrow Y$. Since k is action preserving, we have

$$\forall x \in X(\mu(m, x) = \lambda(m, x)).$$

Then, $\chi_k : (Y, \mu) \to \Omega$ of k is $\chi_k : Y \to L_A$ defined by

$$\chi_k(y) = \{m \mid \mu(m, y) \in X\}, \forall y \in Y.$$

4.2 Topoi

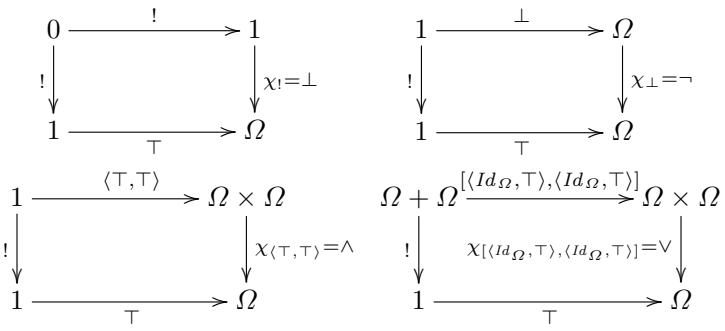
Definition 9 (Topos). *An (elementary)* topos *is a category* $\mathbf{C}$ *that has a terminal object, pullbacks, exponentials and subobject classifier.*

Example 7 (Topos). The $\mathbf{Set}$ category.

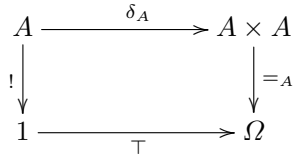
Local Set Theory One of the useful aspects of topos theory, from a logical perspective, is the investigation of the internal logic of a topos by means of a localized language, called local set theory (LST), which is essentially a high-order typed language. This is done by taking any topos as a model of a theory in the language of LST. The interpretation of such a theory in a particular topos provide us with a convenient way of treating the objects of the given topos as set-like entities and the morphisms as function-like relations.

Subobject classifiers allow for the definition of equality, membership relation, and existential and universal quantifiers. In other words, for any topos, subobject classifiers form the semantics of local set theory. As a morphism from $A \times A$ into Ω, $=_A$ is a predicate. LST has a propositional meaning for $\in_A$, $=_A$, $\forall_A$ and $\exists_A$ which are typed (or localized) counterparts for $\in$, $=$, $\forall$ and $\exists$.

Definition 10 (Logical connectives).



Definition 11 (Local identity). *Consider an object A in a topos* $\mathbf{T}$*. Let $\delta : A \to A \times A$ be the diagonal morphism defined as $\langle Id_A, Id_A \rangle$. Local equality in A, denoted $=_A$, is defined by the following subobject classifier pullback, where $=_A$ os the characteristic morphism of δ.*



Definition 12 (Local membership). *Consider a topos* $\mathbf{T}$ *and an object A. Let $eval_A$: $A \times \Omega^A \to \Omega$ be the evaluation morphism provided by the exponential object Ω^A. The*

following instance of the subobject classifier pullback defines $\in_A$.

$$\begin{array}{ccc} \in_A & \xrightarrow{inc} & A \times \Omega^A \\ {\scriptstyle !}\downarrow & & \downarrow{\scriptstyle eval_A} \\ 1 & \xrightarrow[\top]{} & \Omega \end{array}$$

Definition 13 (Local $\forall$). *$R(x^B, y^A)$ is represented by $r : R \to B \times A$.*

$$\begin{array}{ccc} \forall x^A.R & \xrightarrow{\forall_A.r} & B \\ {\scriptstyle !}\downarrow & & \downarrow{\scriptstyle \hat{\chi}_{r_A}} \\ 1 & \xrightarrow{\hat{!}} & \Omega^A \end{array}$$

Definition 14 (Local $\exists$). *$\exists y^A.R(x^B, y^A)$ is $\exists_A \circ \hat{\chi}_r$.*

$$\begin{array}{ccccc} \in_A & \xrightarrow{inc} & A \times \Omega^A & \xrightarrow{\pi_2} & \Omega^A \\ {\scriptstyle !}\downarrow & & & & \downarrow{\scriptstyle \chi_{\pi_2 \circ inc} = \exists_A} \\ 1 & & \xrightarrow{\quad !\quad} & & \Omega \end{array}$$

To conclude this section, we say that a category is *locally small* whenever $Hom(A, B)$ is a *set*, for any A and B in the category.

Theorem 1 (Fundamental theorem of Topoi). *Let $\mathbf{C}$ be a locally small category. $\mathbf{Set}^{\mathbf{C}}$ is a topos.*

The category $\mathbf{Set}^{\mathbf{C}}$, when $\mathbf{C}$ is a pre-order, is naturally interpreted as sets varying according $\mathbf{C}$. The pre-order works as a temporal structure over each set evolves. When $\mathbf{C}$ is more than a pre-order category, sometimes it is possible to see a kind of topology on any object A induced by the morphism with co-domain A. In this case, we have a temporal structure induced by this topology. Anyway, in some cases, $\mathbf{Set}^{\mathbf{C}}$ is naturally equivalent to a category of dynamic systems. Since discrete dynamical systems can be seen as a semantics for computing processes, the use of the above functorial category in providing examples for non-standard model of computing is justified.

4.3 M-Set is a topos

Definition 15 (Left ideal of M). *A set $B \subseteq A$ is a left ideal of M if it is closed under left-multiplication, that is, if $m * b \in B$ whenever $b \in B$ and $m \in A$.*

Theorem 2 (M-Set). **M-Set** *is a topos where each object is a M-set. A morphism $f : (X, \lambda) \to (Y, \mu)$ is an action-preserving (or equivariant) function $f : X \to Y$ such that, for all $m \in A$ and $x \in X$,*

$$f(\lambda(m, x)) = \mu(m, f(x)).$$

Proof. **M-Set** is a category. Arrow composition is functional composition. The terminal object $\mathbf{1} = (\{0\}, \lambda_0)$ such that $\forall m(\lambda_0(m, 0) = 0)$. The product of (X, λ) and (Y, μ) is $(X \times Y, \delta)$ where $\delta_m = \lambda_m \times \mu_m : X \times Y \to X \times Y$. The subobject classifier of **M-Set** is $\Omega = (L_A, \omega)$ where L_A is the set of left ideals in M and $\omega : A \times L_A \to L_A$ such that $\omega(m, B) = \{n \mid n * m \in B\}$. The morphism $\top : \mathbf{1} \to \Omega$ is the function $\top : \{0\} \to L_A$ with $\top(0) = A$. (Function $\top$ selects the largest left ideal A of M.) Given (X, λ) and (Y, μ) we define the exponential

$$(Y, \mu)^{(X,\lambda)} = (E, \sigma)$$

where E is the set of equivariant maps f to the function $g = \sigma_m(f) : M \times X \to Y$ given by $g(n, x) = f(m * n, x)$. The evaluation arrow $eval : (E, \sigma) \times (X, \lambda) \to (Y, \mu)$ has $eval(f, x) = f(e, x)$. Then given an arrow $f : (X, \lambda) \times (Y, \mu) \to (Z, \nu)$, the exponential adjoint $\hat{f} : (X, \lambda) \to (Z, \nu)^{(Y,\mu)}$ takes $x \in X$ to the equivariant map $\hat{f}_x : M \times Y \to Z$ having $\hat{f}_x(m, y) = f(\lambda_m(x), y)$. □

5 Categorifying Refinement

5.1 Refinements are monomorphisms

The process of categorification gives categorial interpretations to set-theoretic concepts. Membership $x \in A$, for instance, is categorified as $x : 1 \to A$ where x is identified with the function $\bullet_x$ from $\{\bullet\}$ in A such that $\bullet_x(\bullet) = x$. The formula $\forall x_1, x_2 \in A(f(x_1) = f(x_2) \Rightarrow x_1 = x_2)$, that expresses that function $f : A \to B$ is injective, in **Set**, is categorified as

$$\forall x_1, x_2 : 1 \to A(f \circ x_1 = f \circ x_2 \Rightarrow x_1 = x_2). \tag{11}$$

It should be noted that the equality $x_1 = x_2$ is an equality between morphisms. In Local Set Theory, it could be expressed as

$$Mono(f^{B^A}) \Leftrightarrow \forall h^{A^C} \forall g^{A^C} [((f \circ h) =_{B^C} (f \circ g)) \Rightarrow (h =_{A^C} g)], \tag{12}$$

where *Mono* stands for monomorphism, a generalization of injective morphisms in **Set**, f^{B^A} is a variable for morphisms $A \to B$. Variables h and g range over morphisms $C \to A$. Note that equality is also properly typed.

To categorify refinement, understood as subautomata relation (see Definition 2), first we need to recall that every automaton is an M-set (see Section 3). Now, every M-set is an object in the category **M-Set** whose arrows are M-set homomorphisms. Therefore, the subautomaton relation is represented categorically as a monomorphism (injective morphism in **Set**) between two M-sets.

In Example 1, the fact that *DFA* A is a subautomaton of C is captured by an injective function f that includes de set of states of A in C, identifies the initial state, includes de set of final states of A in C and the transition relation. Function f is a monomorphim in **M-Set**.

5.2 Lean and Topoi

As we have seen in Section 4.2, every topos has an internal logic or its local set theory. However, *intuitionistic logic is a common denominator to them all* [9, Ch. 8]. Per Martin-Löf devised a calculus for Intuitionistic Type Theory (ITT) that recently received an incarnation in the Lean [2] proof assistant. Listing 1 shows the Lean specification for the exponential object of a topos. Essentially, every structure represents a "data type": in our case, monoids, M-sets, categories and topoi. Structures may be parameterized and have many properties. For instance, the `elementary_topos` structure, following Definition 9, has an exponential object B^A, given objects A, B and C, if there exists morphisms $eval_{AB} : B^A \times A \to B$ (represented by ev in the specification), $g : C \times A \to B$, there exists a unique morphism $curry\_g : C \to B^A$ such that $eval_{AB} \circ (curry(g) \times id_A) = g$. Variables such as p_ba_a denote the product of objects ba and a and $prv\_p\_c\_a$ denotes a proof that there exists the product $C \times A$. Constructor hom builds a homomorphism from two objects.

```
1 structure elementary_topos (ob : Type) extends category ob :=
2 ...
3     (exp : ∀ {a b c ba p_ba_a p_c_a : ob}(g : hom p_c_a b)(curry_g : hom c ba)(ev : hom p_ba_a b)
4               (prv_p_ba_a : ob_prd ba a p_ba_a) (prv_p_c_a : ob_prd c a p_c_a),
5                (comp ev (hom_prd (curry_g) (ID a) prv_p_c_a prv_p_ba_a)) = g)
6 ...
```

Listing 1: Lean specification for exponential object

On a technical note, one of the reasons for choosing Lean over Coq [21], a very mature proof assistant for ITT, is that Lean implements rules

$$\frac{r \in I(A,a,b)}{a = b \in A}\ I\text{-}intro \qquad \frac{a = b \in A}{c \in I(A,a,b)}\ I\text{-}elim \qquad \frac{c \in I(A,a,b)}{c = r \in I(A,a,b)}\ I\text{-}eq$$

from Martin-Löf's type theory that allows us for to go beyond the propositional treatment of equality, that is, reasoning with reflexivity, transitivity, and symmetry, by treating *equality as a type*. (The type I in the rules above.) In programming jargon, it gives a formal infrastructure to *overload* equality and reason about it, so "we can not only have the cake but it too." This feature appears to be quite important while defining the type for morphisms in a category as in Category Theory we can not talk about object equality, only about morphism equality.

6 Final Remarks

Category Theory is an appropriate framework to establish relations among concepts that are perhaps in very different spheres of human knowledge. Topoi is a theory that considers a particular kind of category. In one of its axiomatic definitions, a topos has a terminal object, all pullbacks, an exponential and a subobject classifer. Moreover, a topos has an internal logic that can be used to express properties local to it, called Local Set Theory.

Even though Topoi has been used to express relations among specifications, it appears that its internal logic has not. This is the scope of this work: we propose to perform such a study and evolve it into other relations among specifications, assuming

an automata semantics for them. Such a topos would be a unifying model for different automata-based models, such as those for software components like constraint automata [3] or input/output automata [4].

References

1. J.-R. Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.
2. J. Avigad, L. de Moura, and S. Kong. *Theorem Proving in Lean*. Microsoft Research, https://leanprover.github.io/tutorial/tutorial.pdf.
3. C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
4. S. S. Bauer, R. Hennicker, and M. Wirsing. Interface theories for concurrency and data. *Theoretical Computer Science*, 412:3101–3121, June 2011.
5. P. F. Castro and N. Aguirre. Algebraic foundations for specification refinements. In L. Ribeiro and T. Lecomte, editors, *Proceedings of Brazilian Sympposium on Formal Methods (SBMF 2016)*, (To appear.), 2016.
6. A. Cavalcanti and J. Woodcock. ZRC – a refinement calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, 1998.
7. R. Diaconescu. *Grothendieck Institutions*, pages 253–273. Birkhäuser, Basel, 2008.
8. J. Goguen and G. Malcom. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, August 2000.
9. R. Goldblatt. *Topoi — The Categorical Analysis of Logic*. Dover Publications, Inc., 2006.
10. E. H. Haeusler, L. C. Pereira, and P. A. Veloso. Categorification, set theory and finiteness (in portuguese). *Notae Philosophicae Scientiae Formalis*, 2(1):1–21, may 2013.
11. C. A. R. Hoare and J. He. *Unified Theories of Programming*. Prentice Hall College division, January 1998.
12. M. Johnson, D. Naumann, and J. Power. Category theoretic models of data refinement. *Electronic Notes in Theoretical Computer Science*, 225:21–38, January 2009.
13. L. Lima, A. Miyazawa, A. Cavalcanti, M. Cornélio, J. Iyoda, A. Sampaio, R. Hains, A. Larkham, and V. Lewis. An integrated semantics for reasoning about SysML design models using refinement. *Software & Systems Modeling*, pages 1–28, 2015.
14. A. Madeira, M. A. Martins, L. S. Barbosa, and R. Hennicker. Refinement in hybridised institutions. *Formal Aspects of Computing*, 27(2):375–395, 2015.
15. J. Meseguer. General logics. In H.-D. Ebbinghaus, J. Fernandez-Prida, M. Garrido, D. Lascar, and M. R. Artalejo, editors, *Logic Colloquium'87Proceedings of the Colloquium held in Granada*, volume 129 of *Studies in Logic and the Foundations of Mathematics*, pages 275 – 329. Elsevier, 1989.
16. T. Mossakowski, C. Maeder, and K. Lüttich. *The Heterogeneous Tool Set, Hets*, pages 519–522. Springer, Berlin, Heidelberg, 2007.
17. F. Orejas, M. Navarro, and A. Sánchez. Algebraic implementation of abstract data types: a survey of concepts and new compositionality results. *Mathematical Structures in Computer Science*, 6(1):33–67, 03 2009.
18. B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
19. A. Sampaio, J. Woodcock, and A. Cavalcanti. *Refinement in Circus*, pages 451–470. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
20. Y. V. Srinivas and R. Jüllig. *Specware: Formal support for composing software*, pages 399–422. Springer, Berlin, Heidelberg, 1995.
21. The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, https://coq.inria.fr/distrib/V8.5pl2/files/Reference-Manual.pdf, July 2016.

Chu Spaces As a Toy Model For Quantum Mechanics

Maigan S. da S. Alcântara[1], Wilson R. de Oliveira[2], and Thiago D. O. Silva[3]

[1] Centro de Informática, Universidade Federal de Pernambuco, Av. Jornalista Aníbal Fernandes, s/n, Cidade Universitária, 50.740-560 - Recife - PE,
mssa@cin.ufpe.br

[2] Departamento de Estatística e Informática, Universidade Federal Rural de Pernambuco, Rua Dom Manoel de Medeiros, s/n, Dois Irmãos, 52171-900, Recife/PE,
wilson.rosa@ufrpe.br

[3] Departamento de Matemática, Universidade Federal Rural de Pernambuco, Rua Dom Manoel de Medeiros, s/n, Dois Irmãos, 52171-900, Recife/PE,
thiago.dk@gmail.com

Abstract. We investigate the categorical properties of Chu spaces with the purpose of using these spaces as a model for categorical quantum mechanics. Despite the enormous success of the Hilbert space model approach to Quantum Mechanics, alternative finite models, dubbed toy models, has helped to discriminate what are the really important features of quantum mechanics. We assume the reader familiarity with categories and give a very brief introduction to Chu spaces. Then, we show that the Chu spaces category is a symmetric monoidal, cartesian and compact category. We show that there are dagger structures for some particular subcategories.

Keywords: Category Theory, Chu Spaces, Quantum Mechanics

1 Introduction

Category Theory and Quantum Mechanics may seem at first as two completely different scientific fields. But rather not only they have much in common as they are mutually enriched from views and approaches of one over another [4].

At first the theory of categories can be seen as a generalisation of the algebra of functions [7]. In this context, clearly, the main operation on functions is composition. Actually, a category is an abstract structure made of *objects* and *arrows* between the objects with a fundamental property namely the *compositionality of arrows* [4].

Category Theory is a relatively recent theory, created by S. Eilenberg and S. Mac Lane in 1945 [11], as a result of their work in algebraic topology. Since then it has influenced many areas as a revolutionary way of understanding and approach the subject field. Currently the interplay between Category Theory, Quantum Mechanics, and Computer Science constitutes extremely active areas of research.

There are numerous proposal in the literature that quantum mechanics can be expressed using categories instead of the traditional Hilbert space [1]. Specifically, the *dagger symmetric monoidal categories* ($\dagger - SMC$) with base structures, which can describe many characteristics of Quantum Mechanics [5].

Coecke and Edwards [8] explore some concrete examples of symmetric monoidal categories to model important features of quantum mechanics, such as quantum teleportation protocol [8].These authors show two important facts: that the 'toy' model presented in *Spekkens* is an interesting instance of a categorical and quantum axiomatic; and that in **FRel** - category of finite sets and relations with the cartesian product as a tensor - the set of two elements $\{0, 1\}$ comes equipped with two complementary observable.

In this context, our special attention is given to categories that have finite spaces and their particular cases as the Chu spaces. We direct the study to $*-$autonomous categories.

Chu space is a special case of a construction that originally appeared as an appendix in the book [2].

Interest in $*-$autonomous categories comes with the advent of Linear Logic since these categories provide models for Multiplicative Linear Logic (and with the additional assumptions for all Linear Logics) [16].

Chu construction applied to the category **Set**, of sets and functions, was introduced independently (with the name of "games") by [14] and subsequently (under the name Chu spaces) was subjected to a series of papers produced by Pratt and his collaborators [16].

Application Chu spaces have been proposed in a number of areas, including the concurrency, game theory, fuzzy systems and mathematical studies on the Chu construction (in categorical terms) [3].Moreover, it has an effective structure of comonoid [17] to treat base structures.

In this sense, we will explore the category of Chu spaces, its relevance and potential uses in quantum mechanics. Finally, we investigate the potentials in the category Chu_K.

2 Chu Space

Chu spaces provides a simple, universal and well-structured representation to a range of objects in mathematics. They are simple because of being merely a rectangular matrix whose rows represent points, their columns represent dual states, whose entries are drawn from a set K.

Some notions that we will see in this section are based on the course notes given by Vaughan Pratt [16] and the Michael Barr's book [2].

A Chu space $\mathcal{A} = (A, r, X)$ over a set K, consists of a set A of *points*, a set X of *states*, and a function $r : A \times X \to K$. Note that the function $r : A \times X \to K$ can be regarded as a matrix with rows on A, columns on X and values on K, particularly when the sets involved are finite, as it is in our case.

We call a Chu space *normal* when each column is a function from A to K, i.e., $X \subseteq K^A$. We use the abbreviate notation (A, X) with $r(a, x)$ taken to be $x(a)$, each $x \in X$ now being a function $x : A \to K$ [16].

The *category* Chu_K has as objects Chu spaces (A, r, X) over K, and as morphisms

$$(f, g) : (A, r, X) \to (B, s, Y)$$

where the pair of functions

$$f : A \to B \quad e \quad g : Y \to X$$

is such that for every $a \in A$ and $y \in Y$, we have:

$$s(f(a), y) = r(a, g(y)).$$

This equation is a primitive form of adjunction, and we call it as the *adjoint condition*. The pair (f, g) will be called adjoint pair. Given the adjoint pairs $(f, g) : (A, r, X) \to (B, s, Y)$ and $(f', g') : (B, s, Y) \to (C, t, Z)$ the composite is given by

$$(f', g') \circ (f, g) = (f' \circ f, g \circ g').$$

This composite is itself an adjunct pair because for all $a \in A$ and $z \in Z$ we have:

$$t(f' \circ f(a), z) = s(f(a), g'(z)) = r(a, g \circ g'(z))$$

So that the following diagram commutes:

$$\begin{array}{ccc} A \times Y & \xrightarrow{f \times id_Y} & B \times Y \\ {\scriptstyle id_A \times g}\downarrow & & \downarrow{\scriptstyle s} \\ A \times X & \xrightarrow[r]{} & K \end{array} \tag{1}$$

The associativity of the composition is inherited from the composition in **Set**.

While the pair $(1_A, 1_X)$ of maps identities, respectively A and X, is the identity of the morphism in (A, r, X), i. e., $id_{\mathcal{A}} : (id_A, id_X) : (A, r, X) \to (A, r, X)$. Their isomorphisms are those morphisms (f, g) in respect to which f and g are both bijections.

There is a series of operations over Chu spaces of practical interest. Operations that come from linear logic [12], an approach carried out by Y. Lafont [14], and process algebras.

The *dual* $\mathcal{A}^\perp$ of a Chu space $\mathcal{A} = (A, r, X)$ corresponds to $\mathcal{A}^\perp = (X, \breve{r}, A)$, where $\breve{r}(x, a) = r(a, x)$.

The *tensor product* $\mathcal{A} \otimes \mathcal{B}$ of two Chu spaces $\mathcal{A} = (A, r, X)$ *and* $\mathcal{B} = (B, s, Y)$ is given by $(A \times B, t, \mathcal{F})$ where $\mathcal{F} \subset Y^A \times X^B$ is the set of all pairs (f, g) of functions $f : A \to Y$ *and* $g : B \to X$ for which $s(b, f(a)) = r(a, g(b))$

for all $a \in A$ and $b \in B$, and $t : (A \times B) \times \mathcal{F} \to K$ is given by $t((a,b),(f,g)) = s(b, f(a))(= r(a, g(b)))$.

Associated with the tensor product is the tensor unit $\mathbf{1}$, namely the space $(*, r, K)$, corresponding to the space with one point and $|K|$ states, where $r(*,k) = k$ to $k \in K$.

However, the notions tensor product and dual seen above are given only to Chu spaces objects. To make it a bi-functor over Chu_K we must include morphisms.

We call $f : A \to B$ *continuous* when it has an adjoint from $B^\perp$ to $A^\perp$, i.e. when there exists a function $g : Y \to X$ making (f,g) a Chu morphism.

Given $(f,g) : \mathcal{A} \to \mathcal{B}$, let $(f,g)^\perp : \mathcal{B}^\perp \to \mathcal{A}^\perp$ defined by (g,f). This suggests the notation $f^\perp = g$.

Given the functions $f : \mathcal{A} \to \mathcal{A}'$ and $g : \mathcal{B} \to \mathcal{B}'$, we define $f \otimes g : \mathcal{A} \otimes \mathcal{B} \to \mathcal{A}' \otimes \mathcal{B}'$ is the function $(f \otimes g)(a,b) = (f(a), g(b))$. When f and g are continuous, $f \otimes g$ is also continuous, indeed $(f \otimes g)^\perp$ from $\mathcal{G}$ to $\mathcal{F}$ (where $\mathcal{G}$ and $\mathcal{F}$ consist respectively of pairs $(h' : A' \to Y', k' : B' \to X')$ and $(h : A \to Y, k : B \to X)$) sends $h' : A' \to Y'$ to $g^\perp \circ h' \circ f : A \to Y$ and $k' : B' \to X'$ to $f_\perp \circ k' \circ g : B \to X$.

Proposition 1. *[16] The tensor is commutative and associative, up to a natural isomorphism: $\mathcal{A} \otimes \mathcal{B} \cong \mathcal{B} \otimes \mathcal{A}$ and $\mathcal{A} \otimes (\mathcal{B} \otimes \mathcal{C}) \cong (\mathcal{A} \otimes \mathcal{B}) \otimes \mathcal{C}$.*

Proposition 2. *[9] The tensor unit behaves as expected, i.e. $\mathcal{A} \otimes 1 \cong \mathcal{A} \cong 1 \otimes \mathcal{A}$ through the obvious pairing of isomorphism $(a, *)$ with a.*

3 Categorical quantum mechanics

The definitions in this section are all based on [7].

A *monoidal category* is a category $\mathbf{C}$ equipped with a bifunctor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, a unit object I, such that for all objects A, B and morphisms f, g, h, k, of the appropriate type, we have

$$(g \circ f) \otimes (k \circ h) = (g \otimes k) \circ (f \otimes h) \quad \text{and} \quad 1_A \otimes 1_B = 1_{A \otimes B} \tag{2}$$

and three natural isomorphisms

$$\alpha_{A,B,C} : A \otimes (B \otimes C) \xrightarrow{\sim} (A \otimes B) \otimes C\,;$$
$$\lambda_A : I \otimes A \xrightarrow{\sim} A;$$
$$\rho_A : A \otimes I \xrightarrow{\sim} A,$$

and for all A, B, C, D, A', B', C' and f, g, h the appropriate type, the following diagrams commute

$$\begin{array}{ccc} A \otimes (B \otimes C) & \xrightarrow{\alpha_{A,B,C}} & (A \otimes B) \otimes C \\ \Big\downarrow{\scriptstyle f \otimes (g \otimes h)} & & \Big\downarrow{\scriptstyle (f \otimes g) \otimes h} \\ A' \otimes (B' \otimes C') & \xrightarrow[\alpha_{A',B',C'}]{} & (A' \otimes B') \otimes C' \end{array} \tag{3}$$

$$\begin{array}{ccc} A & \xrightarrow{\lambda_A} & I \otimes A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle 1_I \otimes f} \\ B & \xrightarrow[\lambda_B]{} & I \otimes B \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{\rho_A} & A \otimes I \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f \otimes 1_I} \\ B & \xrightarrow[\rho_B]{} & B \otimes I \end{array} \tag{4}$$

The condition of associativity, is known as the Mac Lane's pentagon [15] Finally, consistency condition.

$$\begin{array}{ccc} A \otimes B & \xrightarrow{1_A \otimes \lambda_B} & A \otimes (I \otimes B) \\ & {\scriptstyle \rho_A \otimes 1_B} \searrow & \downarrow{\scriptstyle \alpha_{A,I,B}} \\ & & (A \otimes I) \otimes B \end{array} \tag{5}$$

$$\lambda_I = \rho_I \,. \tag{6}$$

A monoidal category is moreover **symmetric** if there is a fourth natural isomorphism

$$\sigma = \{A \otimes B \xrightarrow{\sigma_{A,B}} B \otimes A \mid A, B \in \mathbf{C}\}\,,$$

which also obeys several consistency conditions [4]. The symmetric monoidal category form the basic structure for classical and quantum systems.

In categorical quantum mechanics literature we have the notion that generalizes the adjoint operator between Hilbert spaces, in which we get the structure of the dagger monoidal categories which is a monoidal category $\mathbf{C}$ equipped with a contravariant functor involution $\dagger : \mathbf{C}^{op} \to \mathbf{C}$, which is the identity on objects, satisfying the equation:

$$(f \otimes g)^\dagger = f^\dagger \otimes g^\dagger \,. \tag{7}$$

In detail, this means that it associates each morphism $f : A \to B$ in $\mathbf{C}$ his adjoint $f^\dagger : B \to A$ such that for all $f : A \to B$ and $g : B \to C$, satisfy some properties.

A dagger symmetric monoidal categories ($\dagger-$SMC) is a symmetric monoidal category which also has a dagger structure.

The category **FdHilb** of finite dimensional Hilbert spaces and linear maps [8] has a dagger structure: given a linear map $f : A \to B$, the map $f^\dagger : B \to A$ is its Hermitian adjoint in the usual sense. **FdHilb** is also a dagger symmetric monoidal category where the tensor is the usual tensor product of Hilbert spaces and the tensor unit is the scalar field $\mathbb{C}$.

4 Structures in the category of Chu spaces

We now investigate further structures of Chu_K related to categorical quantum mechanics.

Theorem 3. *Chu_K with the tensor product $\otimes$, is a symmetric monoidal category.*

Proof. We have seen that the tensor product unit is the Chu space with one point and $|K|$ states, denoted by $\mathbf{1}$.

Consider the Chu spaces $\mathcal{A} = (A, r, X), \mathcal{B} = (B, s, Y)$ *and* $\mathcal{C} = (C, t, Z)$. By functoriality of $\otimes$ seen in section 2, we have:

$$(h\otimes k)\circ(f\otimes g)(a,b) = (h\otimes k)\circ(f(a), g(b)) = (h\circ f(a), k\circ g(b)) = (h\circ f)\otimes(k\circ g)(a,b)$$

where $f : \mathcal{A} \to \mathcal{A}', g : \mathcal{B} \to \mathcal{B}', h : \mathcal{A}' \to \mathcal{A}''$ *and* $k : \mathcal{B}' \to \mathcal{B}''$.
The three natural isomorphisms are given as follows:

- For natural isomorphism α, seen the proposition 1 the tensor is associative, so: $\alpha_{\mathcal{A},\mathcal{B},\mathcal{C}} : \mathcal{A} \otimes (\mathcal{B} \otimes \mathcal{C}) \to (\mathcal{A} \otimes \mathcal{B}) \otimes \mathcal{C} : \alpha_{\mathcal{A},\mathcal{B},\mathcal{C}} : (a, (b, c)) = ((a, b), c)$;
  The proposition 2, show that the isomorphism λ and ρ are well defined in Chu. Hence,
- $\lambda_{\mathcal{A}} : \mathcal{A} \to 1 \otimes \mathcal{A} :: \lambda_{\mathcal{A}}(a) = (*, a)$;
- $\rho_{\mathcal{A}} : \mathcal{A} \to \mathcal{A} \otimes 1 :: \rho_{\mathcal{A}}(a) = (a, *)$.

Hence satisfying Eqs. 3 and 4, the same goes for the Mac Lane's pentagon. We omit the such diagrams for objectivity effect. The condition 5 is given by:

$$\begin{array}{ccc} \mathcal{A}\otimes\mathcal{B} & \xrightarrow{1_{\mathcal{A}}\otimes\lambda_{\mathcal{B}}} & \mathcal{A}\otimes(1\otimes\mathcal{B}) \\ & \searrow^{\rho_{\mathcal{A}}\otimes 1_{\mathcal{B}}} & \downarrow{\scriptstyle \alpha_{\mathcal{A},1,\mathcal{B}}} \\ & & (\mathcal{A}\otimes 1)\otimes\mathcal{B} \end{array} \tag{8}$$

This condition follows Propositions 1 and 2. Furthermore, we have:

$$\sigma_{I,\mathcal{A}} \circ \lambda_{\mathcal{A}}(a) = \sigma_{I,\mathcal{A}}(*, a) = (a, *) = \rho_{\mathcal{A}}(a).$$

- For symmetry, we saw in the proposition 1 the tensor is commutative, and so the natural isomorphism σ to the Chu space is :
  $\sigma_{\mathcal{A},\mathcal{B}} :: \mathcal{A} \otimes \mathcal{B} \to \mathcal{B} \otimes \mathcal{A} :: \sigma_{\mathcal{A},\mathcal{B}}(a, b) = (b, a)$.

So that the following diagrams commute

$$\begin{array}{ccccc} \mathcal{A}\otimes(\mathcal{B}\otimes\mathcal{C}) & \xrightarrow{\alpha_{\mathcal{A},\mathcal{B},\mathcal{C}}} & (\mathcal{A}\otimes\mathcal{B})\otimes\mathcal{C} & \xrightarrow{\sigma_{(\mathcal{A}\otimes\mathcal{B}),\mathcal{C}}} & \mathcal{C}\otimes(\mathcal{A}\otimes\mathcal{B}) \\ \downarrow{\scriptstyle 1_{\mathcal{A}}\otimes\sigma_{\mathcal{B},\mathcal{C}}} & & & & \downarrow{\scriptstyle \alpha} \\ \mathcal{A}\otimes(\mathcal{C}\otimes\mathcal{B}) & \xrightarrow[\alpha_{\mathcal{A},\mathcal{B},\mathcal{C}}]{} & (\mathcal{A}\otimes\mathcal{C})\otimes\mathcal{B} & \xrightarrow[\sigma_{\mathcal{A},\mathcal{C}}\otimes 1_{\mathcal{B}}]{} & (\mathcal{C}\otimes\mathcal{A})\otimes\mathcal{B} \end{array} \tag{9}$$

This condition immediately follows from Proposition 1 where show that the tensor is commutative and associative!

Thus, Chu_K is a symmetric monoidal category. □

A category is $*-$ autonomous is a symmetric monoidal category equipped with a *dualizing* object $\perp$.

Note that Chu_K is a category $*-$ autonomous with its dual Chu space $\mathcal{A}^{\perp}$ as the dual object of the space $\mathcal{A}$ and the dualizing object defined as $\perp = \mathbf{1}^{\perp}$.

Moreover, Chu_K is a closed compact category with respect to the tensor product and Chu_K is a Cartesian category with respect to the direct sum $\oplus$ [9].

5 Discussion and other results

Recently, many studies in categorical quantum mechanics has given evidence of the importance of the dagger functor (†) for expressing the essential structures of quantum mechanics in a symmetric monoidal category (for example basis structures, entanglement, etc). And so we arrive at categories with more sophisticated structure, such as *dagger symmetric monoidal categories* †− SMC).

Thus is natural to investigate weather the category Chu_K admits such functor and if it is a †− SMC. We shall see that Chu_K has dagger structure in at least two cases: 1. when the morphisms are bijections (i.e. it is a subgroupoid of Chu_K); and 2. When the points set A and states set X have the same cardinality.

Note that to define a *dagger* functor which is identity on objects, the morphisms have to satisfy the following property:

Let $\mathcal{A} = (A, r, X)$ and $\mathcal{B} = (B, s, Y)$ Chu spaces. Given the morphism

$$f = (f_1, f_2) : \mathcal{A} \to \mathcal{B}$$

with $f_1 : A \to B$ and $f_2 : Y \to X$ such that $s(f_1(a), y) = r(a, f_2(y))$. The adjoint morfismo

$$f^{\dagger} = (f_1^{\dagger}, f_2^{\dagger}) : \mathcal{B} \to \mathcal{A}$$

where $f_1^{\dagger} : B \to A$ e $f_2^{\dagger} : X \to Y$ must satisfy $s(b, f_2^{\dagger}(x)) = r(f_1^{\dagger}(b), x)$.

Remark that this property is related to the functoriality of the candidate to † and uses continuity.

Thus, in search of structure of the dagger functor to Chu_K, we impose some restrictions and we get the following results:

CASE 1: In this first case, we consider that the morphisms between two Chu spaces are bijections, i. e., such morphisms have an inverse.

Theorem 4. *Let $\mathcal{A} = (A, r, X)$ and $\mathcal{B} = (B, s, Y)$ Chu space and $f = (f_1, f_2)$ are Chu morphism between $\mathcal{A}$ and $\mathcal{B}$ where $f = (f_1, f_2)$ are such that f_1 and f_2 are bijections then there $f^{\dagger} = (f_1^{\dagger}, f_2^{\dagger})$ between $\mathcal{B}$ and $\mathcal{A}$ where $f_1^{\dagger} = f_1^{-1}$ and $f_2^{\dagger} = f_2^{-1}$.*

Proof. We want to show that with the restriction that we did, we have:

$$s(b, f_2^{\dagger}(x)) = r(f_1^{\dagger}(b), x) \quad \forall b \in B, x \in X.$$

With the hypothesis that f_1 and f_2 are bijections, given $b \in B$ there is a unique a such that $f_1(a) = b$ and a unique y such that $f_2(y) = x$.

Thus,

$$s(b, f_2^\dagger(x)) = s(f_1(a), f_2^\dagger(x)) = r(a, f_2(f_2^\dagger(x))) = r(a, x)$$

and

$$r(f_1^\dagger(b), x) = r(f_1^\dagger(f_1(a)), x) = r(a, x).$$

□

This result corresponds to a particularity of the category $Chu_K - groupoid$, i.e, a *groupoid* is a category in which each morphism has an inverse.

Notation: $gChu_K$ - the category which has Chu spaces as objects and bijections as morphisms.

Another interesting case is the category **FdUnit** which has finite dimensional Hilbert spaces as objects and unitary operators as morphisms.

A morphism is called *unitary* if $f^\dagger = f^{-1}$. Thus, in $gChu_K$ all our morphisms are unitary.

CASE 2: In this case, we use Lafont's approach[14], which sees Chu space as a generalization of vector spaces.

We know that given a finite dimensional vector space X, its dual space, $X^* = \phi : X \to K$, consists of linear maps from X to its scalars field K. Supposing that these spaces have the same dimension, and thus are isomorphic as vector spaces ($X \equiv X^*$), so there is a bijective linear transformation between them: $T : X \to X^*$. In other words, a bijective linear map is an isomorphism between vector spaces.

We extend the basic notions of linear algebra to Chu space by Lafont [14], rewriting the Chu space $\mathcal{A} = (A, r, X)$ as:

$$\mathcal{A} = (X, \langle \_ | \_ \rangle, X^*),$$

r is considered as the *evaluation braket* $(x, f) \mapsto \langle x|f\rangle = f(x)$ which corresponds to a (bilinear) map from $X \times X^*$ to K (with X^* the dual of X).

Thus, a linear map u from $(X, \langle \_ | \_ \rangle, X^*)$ to $(Y, \langle \_ | \_ \rangle, Y^*)$ consists of two maps $u_* : X \to Y$ and $u^* : Y^* \to X^*$ such that:

$$\langle u_*(x)|g\rangle = \langle x|u^*(g)\rangle$$

for all $x \in X$ and $g \in Y^*$.

Therefore, to obtain the dagger structure we suppose that there is a bijection between X and its dual X^*, i.e., we assume that they have the same dimension, $X \equiv X^*$.

Lemma 5. *Let $\mathcal{A} = (X, r, X^*)$ Chu space and $A^\perp = (X^*, \breve{r}, X)$ its dual, where $\breve{r}(x, g) = r(g, x)$, for all $x \in X$ and $g \in X^*$. So*

$$\mathcal{A} \simeq \mathcal{A}^\perp \Leftrightarrow X \equiv X^*.$$

Proof. ($\Rightarrow$) If $\mathcal{A} \simeq \mathcal{A}^{\perp}$, then there is a bijective morphism $i : \mathcal{A} \to \mathcal{A}^{\perp}$ consisting of the pair of bijections (i_1, i_2) such that $i_1 : X \to X^*$ and $i_2 : X^* \to X$.

$$\therefore X \equiv X^*.$$

($\Leftarrow$) If $X \equiv X^*$ then there is a bijection $\sigma : X \to X^*$.Thus, there is the inverse $\sigma^{-1} : X^* \to X$ where given $g \in X^*$ There is a unique $x \in X$ such that $\sigma(x) = g$. Thus, there is an isomorphism $i : \mathcal{A} \to \mathcal{A}^{\perp}$.

$$\therefore \mathcal{A} \simeq \mathcal{A}^{\perp}.$$

□

Theorem 6. *Let $h : (X, \langle \_|\_ \rangle, X^*) \to (Y, \langle \_|\_ \rangle, Y^*)$ Chu morfism, $h = (f, f^*)$ where $f : X \to Y$ and $g : Y^* \to X^*$ such that $X \equiv X^*$ and $Y \equiv Y^*$. Then there a dagger structure $h^{\dagger} : (Y, \langle \_|\_ \rangle, Y^*) \to (X, \langle \_|\_ \rangle, X^*)$ with $h^{\dagger} = (f^{\dagger}, g^{\dagger})$ where $f^{\dagger} : Y \to X$ and $g^{\dagger} : X^* \to Y^*$.*

Proof. If $f : X \to Y$, by hypothesis we have to $X \equiv X^*$ and $Y \equiv Y^*$. Hence, there is a morphism $f' : X^* \to Y^*$, with $f'(x^*) = f'(x^*)^* = f(x)$, for all $x^* \in X^*$ and $x \in X$. Analogously, if $g : Y^* \to X^*$ by hypothesis we have to $X^* \equiv X$ and $Y^* \equiv Y$ then there a morphism $g' : Y \to X$ such that $g'(y) = g(y^*)^* = g(y^*)$, for all $y^* \in Y^*$ and $y \in Y$. Taking $f^{\dagger} = g'$ and $g^{\dagger} = f'$ we get the desired dagger structure. □

Basis structure forms an essential part of any quantum category, in the approach of [6] and that several authors axiomatize Quantum Mechanics using *dagger closed compact categories*.

Coeke and Pavlovic [6] points out that the definition of complementary structures in **FdHilb** coincides with the standard one in quantum mechanics and an equivalent algebraic characterization of complementary observables is given by the following theorem:

Theorem 7 ([6]). *In a category with enough points each pair of complementary basis structures forms a (scaled) Hopf bialgebra with trivial antipode.*

Coeke and Pavlovic continue to show that this abstract definition captures most of the complementary observable behavior of quantum mechanical systems [6].

For the particular case in which every morphism is an isomorphism, with our 'involution' functor † defined over the tensor product $\otimes$, we have the structure *comonoid* $(\mathcal{A}, \delta, \epsilon)$ where $\delta : \mathcal{A} \to \mathcal{A} \otimes \mathcal{A}$ is the diagonal map $\delta(a) = (a, a)$ and $\epsilon : \mathcal{A} \to 1$ is constant map $\epsilon(a) = 1$, provides a base structure for the particular subcategories seen in the cases 1 and 2.

Category Theory have been used in Quantum Mechanics earlier than in [6]. For example, [10] proposes the use of *monoidal comonads*. In [18] is proposed the use of *Frobenius monadas* and pseudomonads. All based on monoidal category, which Chu_K is. For a basic introduction to this part of the Categorical Quantum

Mechanics field we suggest reading the book [19]. We do not develop this line here which is much more complex and general with applications in Topological Quantum Field Theory (see for example [13]). There is much to explore and this is what we intend to do in the continuations of this work.

References

[1] S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. *In: Proceedings of 19th IEEE conference on Logic in Computer Science*, page 415–425., 2004.

[2] M. Barr. **-Autonomous categories with an Appendix by Po-Hsiang Chu*, volume 752 of *Lecture notes in mathematics, 752*. Springer-Verlag, 1979.

[3] Michael Barr. The separated extensional Chu category. *Theory and Applications of Categories*, 4(6):137–147, 1998.

[4] B. Coecke. *New Structures for Physics*, volume 813. Springer-Verlag Berlin Heidelberg, 2011.

[5] B. Coecke, E. O. Paquette, and D. Pavlovic. Classical structures and quantum structures. In S. Gay and I. Mackie, editors, *Semantic Techniques in Quantum Computation and Iformation*. Cambridge University Press, 2008.

[6] B. Coecke and D. Pavlovic. *Quantum measurements without sums*. In: Mathematics of Quantum Computing and Technology, G, 2007.

[7] Bob Coecke. Introducing categories to the practicing physicist. In *What is category theory*, pages 45–74, 2006.

[8] Bob Coecke and Bill Edwards. Toy quantum categories (arXiv:0808. *Electronic Notes in Theoretical Computer Science*, 2008.

[9] Maigan S. da S. Alcântara. Espaço de chu como modelo para mecânica quântica. Master's thesis, Universidade Federal Rural de Pernambuco, Recife - Brazil, 2016.

[10] Brian Day and Ross Street. Quantum categories, star autonomy, and quantum groupoids. In *in" Galois Theory, Hopf Algebras, and Semiabelian Categories", Fields Institute Communications 43 (American Math. Soc.* Citeseer, 2004.

[11] S. Eilenberg and Mac Lane. *General theory of natural equivalences*, volume 58. Transactions of the American Mathematical Society, 1945.

[12] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci. (TCS)*, 50:1–102, 1987.

[13] Joachim Kock. *Frobenius algebras and 2-d topological quantum field theories*, volume 59. Cambridge University Press, 2004.

[14] Yves Lafont and Thomas Streicher. Games semantics for linear logic. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 43–50. IEEE, 1991.

[15] S. MacLane. *Categories for the Working Mathematician*. 2nd edition. Springer-Verlag, 1998.

[16] V. R. Pratt. Chu spaces: Notes for school on category theory and applications. Technical report, University of Coimbra, Coimbra, Portugal, July 1999.

[17] Vaughan R. Pratt. Comonoids in chu: a large Cartesian closed sibling of topological spaces. *12 pp. (electronic), in CMCS'03: Coalgebraic Methods in Computer Science, Proceedings of the 6th Workshop held in Warsaw*, 82(1):(electronic, 2003.

[18] Ross Street. Frobenius monads and pseudomonoids. *Journal of mathematical physics*, 45(10):3930–3948, 2004.

[19] Ross Street. *Quantum Groups: a path to current algebra*, volume 19. Cambridge University Press, 2007.

Tool Support for Formal Component-based Development

D. I. A. Pereira, M. V. M. Oliveira and S. R. R. Silva

Universidade Federal do Rio Grande do Norte – Brazil
dalayalmeida@ppgsc.ufrn.br, marcel@dimap.ufrn.br, sarahraquelrs@gmail.com

Abstract. In previous work we have presented a CSP based approach for developing component-based asynchronous systems, $\mathcal{BRIC}$, which guarantees deadlock freedom by construction. It uses CSP to specify the constraints and interactions between the components to allow a formal verification of the composition's behaviour. In this work we present a tool that automates the verification of component composition by automatically generating and checking the side conditions imposed by the approach. Besides this, the tool also includes a support to $\mathcal{BRICK}$, an optimisation of $\mathcal{BRIC}$, that enriches the components with metadata containing additional useful information, decreasing the complexity of the composition verifications.

Keywords: Component-Based Systems, CSP, Automation

1 Introduction

The use of increasingly complex applications is demanding a greater investment of resources in software development processes. Component-based System Development (CBSD) has emerged [7] as a promising approach for mastering this complexity. In this paradigm, the system is divided into independent pieces of software (components) that can interact and communicate with each other, yielding a final more complex system. A component is a composition unity with contractually specified interfaces and with only explicit context dependencies.

Although CBSD has improved the quality of final products and the organisation of the development process, it lacks formalisation, which is still a big source of problems specially for reliable systems. In order to improve reliability, Formal Methods arise in the development cycle, solving some of the problems and proving to be an interesting development approach for critical systems.

Communicating Sequential Processes [6] (CSP) is a formal notation used to model concurrent and reactive applications where processes interact with each other exchanging messages. The use of CSP allows us to identify problems such as deadlock (when two or more processes are blocked because they are waiting the execution or resources held by each other). CSP has a set of tools that facilitate its use like, for instance, the model-checker FDR3 [3].

In [10, 11], Ramos presented an approach, called $\mathcal{BRIC}$, for the trustful and systematic development of component-based systems for CSP models. In $\mathcal{BRIC}$,

component composition is achieved using four predefined rules that impose restrictions on the basic components. Once the restrictions are satisfied, deadlock-freedom is ensured by construction. Besides the $\mathcal{BRIC}$ strategy, Ramos developed an extension, $\mathcal{BRICK}$, that inserts metadata inside the components as a way to decrease the number and the complexity of the verifications when composing components.

The use of these approaches may demand too much effort on specifying components, their compositions and still guarantee the correctness of the whole specification, which can make their practical application too complex and cumbersome. This paper presents a tool, *BTS* ($\mathcal{BRIC}$ Tool Support), which provides a simpler way to create and compose $\mathcal{BRIC}$ components. More importantly, it provides a complete analysis of the components and their compositions, making the development process safer and more efficient. The tool also considers the extension of $\mathcal{BRIC}$, $\mathcal{BRICK}$, allowing the user to include metadata to the components and decreasing even more the verification costs.

In Section 2 we introduce CSP. $\mathcal{BRIC}$ and $\mathcal{BRICK}$ are described in Section 3. *BTS* is presented in Section 4. Its evaluation is described in Section 5. Finally, we draw our conclusions and discuss about related and future work in Section 6.

2 CSP

CSP is one of the most well established formalisms for describing and analysing concurrent systems. It is used to model applications where independent components called processes interact with each other and with the outer world exchanging messages (events).

In CSP, the most basic processes are `SKIP` and `STOP`; the former successfully terminates the execution, and the later deadlocks. The prefixing `a -> P` is initially able to perform the simple event `a`, after which it will behave like `P`. The prefixing operator can also be used to denote directional communication. The process `c!v -> P` sends the value `v` via channel `c` and behaves as `P` afterwards; and the process `c?x -> P` receives a value through channel `c` and assigns it to the implicitly declared variable `x`, which can then be referred to in the subsequent behaviour `P`. Two processes can be composed in interleave. In `P ||| Q`, the processes `P` and `Q` execute concurrently, but they do not synchronise on any event. It is also possible to compose two processes in parallel synchronising in a specific set of event `cs`. In `P [| cs |] Q`, the processes `P` and `Q` are executed concurrently and synchronise on the events in `cs`.

There are several well-established semantic models of CSP, one of them is the traces model ($\mathcal{T}$) [12]. The set $traces(P)$ contains all possible sequences of events in which P can engage. In order to verify the truthfulness of a property on FDR3 it is possible to define assertions like `assert P :[deadlock free]`, that, in this case, checks if the process `P` holds the deadlock freedom property.

3 BRIC

BRIC [11] is a method for trustful and systematic development of component based systems. It describes components as contracts and imposes restrictions to component compositions in order to guarantee the safety of the final results. BRIC provides four composition rules (two binary rules and two unary rules), each one with specific well defined conditions for valid applications. The component contract definition is presented below:

Definition 1 (Component contract). *A component contract* $Ctr : \langle \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{C} \rangle$ *comprises its behaviour* $\mathcal{B}$*, which is described as a restricted form of CSP process, I/O process, described below, a set of channels* $\mathcal{C}$*, a set of data types* $\mathcal{I}$*, and a total function* $\mathcal{R} : \mathcal{C} \to \mathcal{I}$ *from channels to their types.*

We use $\mathcal{B}_{Ctr}$, $\mathcal{R}_{Ctr}$, $\mathcal{I}_{Ctr}$ and $\mathcal{C}_{Ctr}$ to denote the elements of the contract Ctr. The behaviour $\mathcal{B}_{Ctr}$ is an I/O process, that is: whenever $c.x \in \alpha_P$, then c is either an input or an output channel (α_P denotes the set of events that P can communicate); P has infinite traces (but finite state space); P is divergence free, P is input deterministic, that is, after every trace of P, if a set of input events of P may be offered to the environment, they may not be refused by P after the same trace; P is strongly output decisive, that is, all choices (if any) among output events on a given channel in P are internal.

Usually, a component is defined once and reused multiple times, and in multiple different contexts. In this work, we represent these contexts as a set of channels, since channels represent interaction points of the component, and each channel is used to communicate with a single component in the environment. So, replacing the channels of a component contract by another set means that it supposedly interacts with another environment.

The interactions between two contracts in a composition must be asynchronous mediated by a (possibly infinite) bi-directional buffer ($BUFF_{IO}$). The asynchronous binary composition between the contracts Ctr_1 and Ctr_2 on the channels *ic* and *oc* is represented by $Ctr_1 \langle ic \rangle \asymp \langle oc \rangle Ctr_2$ and the asynchronous unary composition of Ctr_1 on the channels *ic* and *oc* is represented by $Ctr_1 \asymp \Big|_{\langle oc \rangle}^{\langle ic \rangle}$.

The first composition rule is Interleave, which aggregates two independent entities such that will not communicate with each other.

Definition 2 (Interleave composition). *Let* Ctr_1 *and* Ctr_2 *be two component contracts, such that* $\mathcal{C}_{Ctr_1} \cap \mathcal{C}_{Ctr_2} = \emptyset$*. Then, the interleave composition of* Ctr_1 *and* Ctr_2*, namely* $Ctr_1 \,[\!|||\!]\, Ctr_2$*, is given by* $Ctr_1 \,[\!|||\!]\, Ctr_2 = Ctr_1 \langle\rangle \asymp \langle\rangle Ctr_2$*.*

In this rule, components do not share any channel, so no synchronisation is performed. It is a particular kind of composition that involves no communication.

The second rule is based on the traditional way to compose two components, attaching two components connecting two channels, one from each component. In order to attach the channels, protocols must have been defined. A *protocol* is an I/O process that inputs solely by a unique channel and outputs solely

by a unique channel. The proviso of strong compatibility between two channels ensures that the outputs of each process is always accepted by the other process. Hence, no information generated (an output) by a process is leaked; all of them are accepted by its peer in the communication.

Definition 3 (Communication composition). *Let Ctr_1 and Ctr_2 be two component contracts, and ic and oc two channels, such that: $ic \in \mathcal{C}_{Ctr_1} \wedge oc \in \mathcal{C}_{Ctr_2}$, $\mathcal{C}_{Ctr_1} \cap \mathcal{C}_{Ctr_2} = \emptyset$, and; $\mathcal{B}_{Ctr_1} \upharpoonright \{ic\}$ and $\mathcal{B}_{Ctr_2} \upharpoonright \{oc\}$ are strong compatible. Then, the communication composition of P and Q, namely $Ctr_1[ic \leftrightarrow oc]Ctr_2$, via ic and oc is defined as follows: $Ctr_1[ic \leftrightarrow oc]Ctr_2 = Ctr_{1\langle ic\rangle} \asymp_{\langle oc\rangle} Ctr_2$.*

$\mathcal{BRIC}$ also provides unary compositions, which enables building systems with cyclic topologies, assembling two channels of the same component. As a result, due to the existence of possible cycles, new conditions are required to preserve deadlock freedom. The unary compositions rules are: feedback and reflexive.

Definition 4 (Feedback composition). *Let Ctr be a component contract, and ic and oc two communication channels, such that $\{ic, oc\} \subseteq \mathcal{C}_{Ctr}$ are independent in Ctr, and $\mathcal{B}_{Ctr} \upharpoonright ic$ and $\mathcal{B}_{Ctr} \upharpoonright oc$ are strong compatible. Then, the feedback composition of Ctr, namely $Ctr[oc \hookrightarrow ic]$, hooking oc to ic, is defined as follows: $Ctr[oc \hookrightarrow ic] = Ctr \asymp\big|_{\langle oc\rangle}^{\langle ic\rangle}$.*

The feedback composition represents the simpler unary composition case, where two channels of the same component are assembled, but do not introduce a new cycle [10]. This is achieved by the proviso of *decoupled channels*. A channel ch_1 is independent (or decoupled) of a channel ch_2 in a process when any communication of ch_2 does not interfere with the order of events communicated by ch_1. It means that they are offered to the environment independently.

The reflexive composition rule deals with more complex systems that indeed present cycles of dependencies in the topology of the system structure. This rule connects dependent channels, which may introduce undesirable cycles of dependencies among the communication of events in the system. In order to compose components by the reflexive rule, the component's behavior must be buffering self-injection compatible, which is very similar to the notion of strong compatibility, except for the fact that we do not compare the communication between two simple processes but between events of the same process [10].

Definition 5 (Reflexive composition). *Let Ctr be a component contract, and ic and oc two channels, such that: $\{ic, oc\} \subseteq \mathcal{C}_{Ctr}$, and $\mathcal{B}_{Ctr} \upharpoonright \{ic, oc\}$ is buffering self-injection compatible, then, the reflexive composition is defined as $Ctr[ic \leftrightharpoons oc] = Ctr \asymp\big|_{\langle oc\rangle}^{\langle ic\rangle}$.*

$\mathcal{BRICK}$ enriches components with metadata in a way to decrease the number of verifications made when composing them, since some properties of the new components can be predicted (using their parents) and maintained on the metadata. *Context Processes* is one metadata example that represents all the

possible communications between a protocol P and another process compatible with it, which allows us to restrict proofs concerning communication via a specific protocol. The enriched component contract is defined below.

Definition 6 (Enriched component contract). *Let Ctr be a component contract, and $\mathcal{K}$ a metadata derived from its elements. An enriched component contract that includes Ctr is represented by $Ctr : \langle \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{C}, \mathcal{K} \rangle$, where $\mathcal{K} : \langle Prot, CTX, DProt, Dec \rangle$ comprises, a relation from channels to protocols Prot, a relation from channels to context processes CTX, a relation from channels to dual protocols DProt and a relation between channels Dec.*

Given a protocol P, a *dual protocol DP* is a deadlock-free protocol such that $inputs(P) = outputs(DP)$, $outputs(P) = inputs(DP)$, $traces(DP) = traces(P)$.

The metadata is recalculated in every composition from the components parents in a way the user will never have to describe them again. The use of metadata decreases the number and the complexity of verifications, since the user is giving more information about the components. Instead of having to check compatibility among port-protocols in a process P, we check this on port-protocols within the metadata. Instead of verifying that two channels are decoupled in the same process, we verify it directly on relations between channels within the metadata. In this way, we perform lightweight verifications.

The metadata also predicts which channels are decoupled in the new composed component, saving this information on the metadata. This information is useful for feedback compositions, in order to avoid the verifications of the independency between channels. However, the resulting metadata may be incomplete, which means that some scenarios can be not predicted. In this case, when composing two components, if the system does not find the metadata needed to make verifications, it will use $\mathcal{BRIC}$ rules of verification instead of $\mathcal{BRICK}$ rules. The advantages of using metadata are described in Section 5.

4 *BRICK* Tool Support

The $\mathcal{BRICK}$ approach tend to be too much exhaustive and complex when used in practice. In order to make it applicable, we developed *BTS*, a tool that assists the systematic and trustful development of component-based system. The tool generates part of the specification automatically, and verifies the whole specification in order to guarantee deadlock-freedom based on the $\mathcal{BRICK}$ strategy.

Developed in Java, *BTS* runs on Windows and Linux and its architecture is composed by 5 main modules: a user interface (GUI), a controller (which intermediates the interaction between the GUI and the other modules), a logic model (which specifies and coordinate the basic structures), a specification module (creating the specifications based on the data the user inserted on GUI) and a FDR3 communication module (which communicates with FDR3 and process the results). The architecture of BTS is presented on Figure 2.

Using *BTS*, the user follows a sequence of steps to specify a system. The home screen (Figure 1) shows four lists to which we may add elements. They contain

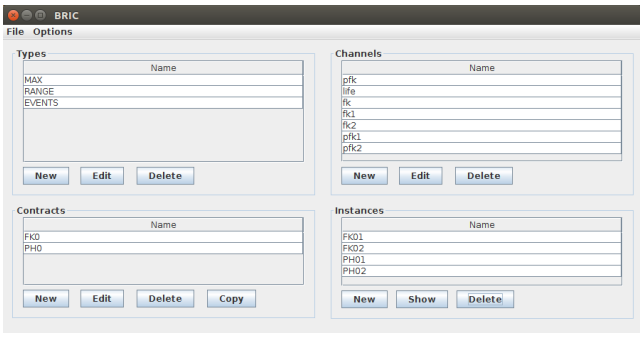


Fig. 1. The home screen of the *BTS* tool

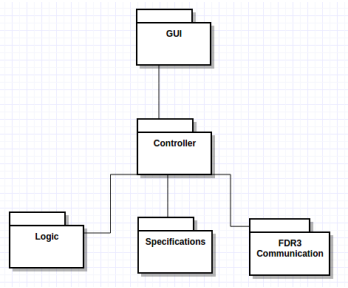


Fig. 2. The architecture of BTS

types, channels, contracts and instances of the specification. The definition of a type requires a name and its definition. A type can be defined as integer, boolean, interval of integers or datatype. The definition of a channel requires its name and its communication types, which must be chosen from the existing list of types. After defining types and channels, the user is ready to define contracts.

The contract definition first requires its name and its CSP behaviour. The contract channels can be chosen from the list of channels. It is also possible to define them as input or output and to define the communication channels (specifying events from the channels). The contract screen (in Figure 3) also allows the definition of metadata information (protocols, dual protocols, context processes and decoupled channels) by double-clicking on a cell on the table.

The definition of the contract behaviour, protocols, dual protocols, and context processes must be made as CSP processes. The decoupled channels are defined by pairing channels from the list of the existing channels. All the metadata information are checked during definition to guarantee their correctness. Some verification requires internal interaction with FDR3. If a metadata is not defined correctly, it will not be used when composing components. The behavior definition and decoupled channels definition screens are presented in Figure 4.

Before the user finishes the contract definition, he must verify if the behavior of the contract is an I/O process. This verification is made by FDR3 and the

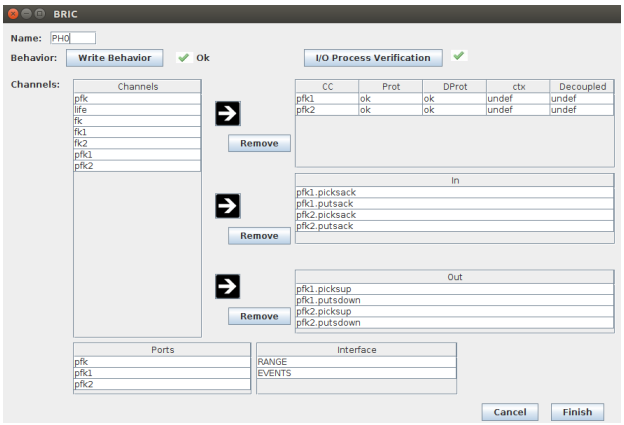


Fig. 3. Contract definition screen

user just needs to press a button. Internally, *BTS* automatically generates the CSP scripts that contains the required specification and assertions, interacts with FDR3 retrieving the verification results and, in case of definition errors, a screen will show what is wrong based on the FDR3 results.

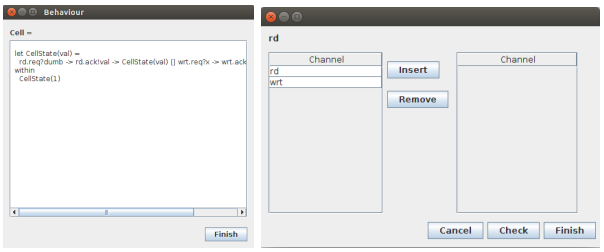


Fig. 4. Behavior and decoupled channels definition screens, respectively

The contract instantiation screen allows us to rename every channel and to define the name of the new instance. Internally, *BTS* verifies that the instances are indeed a valid component contract. After instantiating the components, *BTS* allows users to compose these instances. This simply requires the selection of the components that will be composed, the channels on which they will communicate (except for Interleaving) and the kind of composition. In order to verify the validity of the composition, the tool automatically makes the definition of the assertions, the interaction with FDR3 and the analysis of the results. *BTS* returns an error screen that describes any possible errors in the composition or adds the newly created component to the list of components instances, removing the composing contracts from this list.

The assertions generated and verified by FDR3 to verify deadlock-freedom were based on the work presented in [9], in which we define CSP assertions that correspond to the composition rules side conditions defined in [10]. In [9], we also describe the assertions that can be discarded when using metadata. This optimisation is also incorporated in *BTS*.

The use of metadata in *BTS* anticipates the verification of some of the compositions side conditions to the verification of the contract definition. Besides, some assertions that were required to verify the correctness of the compositions are verified earlier in the contract definition and, hence, they no longer need to be verified. *BTS* also recalculates the new metadata of a component based on the metadata of the composing components. Furthermore, composing components can only be instantiations of previously defined components contracts. The verification of their metadata, however, are normally made only once. If a Contract P that uses metatada is instantiated to contracts P_1, P_2 and P_3 some verifications are only made in P's metadata. If no metadata is used, all verifications are made during composition for each contract instance involved.

5 Evaluation

The tools initial evaluation was achieved based on two case studies. The first one is the dining philosophers described in [11]. The second case study was a ring buffer described in [9, 14]. The results of this evaluation can be found in Table 1.

Table 1. Evaluation details of the *BTS* tool

Evaluation	Dining Philosophers	Ring Buffer
Contracts	2	2
Instances created	6	4
Verifications of protocols metadata	4	8
Verifications of Dec. channels metadata	0	1
Interleave compositions	4	2
Communications compositions	1	1
Feedback compositions	8	4
Reflexive compositions	1	1
Assertions sent to FDR3	344	270
Lines of CSP specification	500	501
System verification time (ms)	4632	7593

Using *BTS*, the definition of types, channels, interfaces, contracts and instances were made in an automatically way with the use of an interface to guide the user. The verification of these contracts and their instances using FDR3 was transparent to the user. All compositions rules were used in these examples and verified automatically, using FDR3, by *BTS*. We have also included metadata into the contracts and used the optimised version of the development approach.

The use of metadata considerably reduced the number of assertions. In the dining philosophers, we had four protocol verifications during the basic contracts definition (two for each contract). If the original $\mathcal{BRIC}$ approach were applied, this verification would be made only on instances, which would increase the number of verifications needed to twelve, two for each instance. The same applies to the protocols and dual protocols of both examples. It was also not necessary to define decoupled channels, but, after each composition, recalculating was executed to predict which channels were decoupled. However, the recalculation after the communication composition in the Ring Buffer example was not complete, and the first feedback composition needed to be verified using the original approach. This switch of rule application from the optimised version to the corresponding original one was also transparent to the user.

More than 600 assertions were generated and sent to FDR3 automatically. *BTS* also received the FDR3 results and presented them, hiding the CSP details that may have invalidated the compositions. These details, however, are easily accessible in case the user requires them. Using these assertions, the tool was able to automatically prove the absence of deadlocks in both examples. Furthermore, over 1000 lines of specification were automatically created by *BTS* after the last compositions. These numbers clearly demonstrate the amount of exhaustive work that *BTS* automatises, which would be made manually otherwise.

The verification of all assertions was made ina a computer Intel(R) Core(TM) i7-3537U CPU 2.00GH, 8Gb RAM, Windows Embedded 8.1 Industry Pro 64bits. The overall time of this verification is presented in the last line of the table.

6 Conclusion

In this work we have developed a tool that automates the systematic construction of trustworthy component-based systems, which makes use of CSP and FDR3 to generate the specifications of the system and verify them automatically. The tool was evaluated by two case studies, generating and verifying all the specifications in a transparent manner to the user.

BTS is the first tool that automates the $\mathcal{BRICK}$ approach, however, some tools have been created to automate formal approaches for specifying component based systems. Some of the existing tools are [13], [15] and [2], which use existing approaches ([1], [5] and [4], respectively) for creating and verifying component based developments. The $\mathcal{BRICK}$'s limitation is the CSP expressiveness limit, which allows us to describe a bigger variety of systems when compered to other approaches. [1], [5] and [4] do not use protocols to alleviate the costs of verification. Furthermore, [1] and [5] are applied only to embedded and distributed systems and [4] does not present efforts to allow reuse of components. $\mathcal{BRICK}$ is not limited to embedded or distributed systems, and the use of protocols and the reuse of components are some of the advantages of this approach.

The work presented in [8] presents an optimisation to $\mathcal{BRICK}$ that makes use of behavioral patterns to reduce the verification costs during composition. The introduction of this approach to *BTS* is in our near future research agenda.

References

1. Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The save approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, 2007.
2. Nikola Beneš, Luboš Brim, Ivana Černá, Jiří Sochor, Pavlína Vařeková, Barbora Zimmerová, et al. The coin tool: Modelling and verification of interactions in component-based systems. pages 221–225, 2008.
3. A.W. Roscoe Thomas Gibson-Robinson Philip Armstrong Alexandre Boulgakov. FDR3 — A Modern Refinement Checker for CSP. In Erika brahm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
4. Luboš Brim, Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. In *ACM SIGSOFT Software Engineering Notes*, volume 31. ACM, 2005.
5. Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.
6. Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
7. M Douglas McIlroy, JM Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*. sn, 1968.
8. M. V. M. Oliveira, P. Antonino, R. Ramos, A. Sampaio, A. Mota, and A. W. Roscoe. Rigorous Development of Component-Based Systems using Component Metadata and Patterns. *Formal Aspects of Computing*, pages 1 – 68, 2016. The original publication is available at www.springerlink.com.
9. M. V. M. Oliveira, A. C. A. Sampaio, P. R. G. Antonino, J. D. Oliveira, M. C. Filho, and J. Bryans. Compositional Analysis and Design of CML Models. Technical Report D24.4, COMPASS Deliverable, 2014.
10. Rodrigo Ramos, Augusto Sampaio, and Alexandre Mota. Systematic development of trustworthy component systems. In *Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009.
11. Rodrigo Teixeira Ramos. *Systematic Development of Trustworthy Component-based Systems*. PhD thesis, PhD thesis, Center of Informatics-Federal University of Pernambuco, Brazil, 2011.
12. Andrew William Roscoe, Charles AR Hoare, and Richard Bird. *The theory and practice of concurrency*, volume 1. Prentice Hall Englewood Cliffs, 1998.
13. Séverine Sentilles, Anders Pettersson, Dag Nystrom, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-ide-a tool for design, analysis and implementation of component-based embedded systems. In *Proceedings of the 31st International Conference on Software Engineering*, pages 607–610. IEEE Computer Society, 2009.
14. Sarah Raquel da Rocha Silva. Bts: uma ferramenta de suporte ao desenvolvimento sistemático de sistemas confiáveis baseados em componentes. Master's thesis, Universidade Federal do Rio Grande do Norte, 2013.
15. Ousmane Sy, Rémi Bastide, Philippe Palanque, D Le, and David Navarre. Petshop: a case tool for the petri net based specification and prototyping of corba systems. *Petri Nets 2000*, page 78, 2000.

Non-involutive bilattices

Paulo Maia[1] Umberto Rivieccio[2] Achim Jung[3]

[1] Programa de Pós-Graduação em Matemática Aplicada e Estatística, UFRN, Brazil
[2] Departamento de Informática e Matemática Aplicada, UFRN, Brazil
[3] School of Computer Science, University of Birmingham, UK

Abstract. One of the main intuitions behind bilattices is to view truth values as split into two components, representing respectively positive and negative evidence concerning a given proposition. Since positive and negative evidence do not need to be the complement of each other, this framework allows one to deal with partial as well as inconsistent information. On an algebraic level, this intuition is reflected in the fact that every bilattice can be represented as a special product of two lattices: positive-evidence lattice and the negative-evidence lattice. In principle, such lattices do not need to be related, that is, the domains of positive and negative evidence may coincide. In this work, we look at algebraic structures having a pre-bilattice reduct and a negation operator that is no longer required to be involutive, which we call non-involutive bilattices. Our contribution is threefold: (1) we show that non-involutive bilattices are a general framework that encompass many of the above-mentioned structures: namely, pre-bilattices, bilattices with an involutive negation, bilattices with implication and d-frames; (2) e provide equational presentations for the class of all non-involutive bilattices and the subclasses corresponding to bilattices with an involutive negation, bilattices with implication etc; (3) finally, for each of these we prove a representation theorem that allows us to view any algebra in the class as a bilattice product of two lattices.

1 Introduction

Nuel Belnap [2] gave a famous philosophical justification for considering two orders on truth value spaces, the information order and the logical order. In this respect he suggested that, in addition to the classical logical values *true* and *false*, it would be useful to have values $\top$ and $\bot$ for the information order, corresponding to the situation when there is contradicting information ($\top$) and lack of information ($\bot$).

Belnap's approach was generalized by Matthew Ginsberg [7], who introduced this generalization as a uniform framework for inference in Artificial Inteligence. Since then, the Belnap-bilattice formalism has found a variety of applications in quite different areas from the original ones. Nowadays the interest in bilattices has thus different sources, among others: computer science and A.I. [7], [1], logic programming [6], lattice theory and algebra [11], algebraic logic and topological duality theory [3], [4], [10], [5].

One of the main intuitions behind bilattices is to view truth values as split into two components, representing respectively positive and negative evidence concerning a given proposition. Since positive and negative evidence need not be the complement of each other, this framework allows one to deal with partial as well as inconsistent information. At the algebraic level, this intuition is reflected by the fact that every bilattice can be represented as a special product $L_1 \times L_2$ (called *bilattice product* or *twist-structure*) of two lattices (L_1 being the positive-evidence lattice and L_2 the negative-evidence lattice). In principle L_1 and

L_2 do not need to be related, that is, the domains of positive and negative evidence may not coincide. However, all bilattice-based logics considered in the literature so far (Ginsberg, Fitting, Arieli-Avron) rely on the assumption that L_1 and L_2 are isomorphic. This structural constraint is imposed by the presence of an involutive negation in the logical language, that is a negation that behaves classically in that any proposition φ is equivalent, in the strongest possible sense, to $\neg\neg\varphi$.

In this contribution, we look at algebraic structures having a *pre-bilattice* reduct (see e.g. [4]) and a negation operator that is no longer required to be involutive, which we call *non-involutive bilattices*. We believe these to be natural structures to be considered from the point of view of the the Belnap-Ginsberg original motivation, for there is no reason to assume that the domain of positive and that of negative evidence must coincide. Furthermore, non-involutive bilattices allow us to rigorously formulate a very natural and expected connection between bilattice-based logics and the topological setting of *d-frames* and *bitopological spaces* [9].

We show that non-involutive bilattices are a general framework that encompass many of the above-mentioned structures: namely, pre-bilattices, bilattices with an involutive negation, bilattices with implication [3] and d-frames. We provide equational presentations for the class of all non-involutive bilattices and the subclasses corresponding to bilattices with an involutive negation, bilattices with implication etc. For each of these we prove a representation theorem that allows us to view any algebra in the class as a bilattice product of two lattices and a categorical equivalence for non-involutive bilattices. The key to our generalized product bilattice construction is to consider pairs of lattices L_1, L_2 together with maps $n\colon L_1 \to L_2$, $p\colon L_2 \to L_1$ between them. These maps allow us to turn positive into negative evidence and vice versa, without requiring the two domains to be isomorphic. By imposing additional properties on the maps n and p (e.g. being meet-preserving) we are then able to recover various bilattice-type structures considered in the literature as special cases of our non-involutive bilattices.

This work is a generalization of [8], to which we also refer for further technical details on the product construction of non-involutive bilattices.

The rest of the paper is organized as follows. In Section 2 we introduce some basic concepts of bilattices; Section 3 presents non-involutive bilattices and gives a categorical interpretation for them, while in Section 4 we do the same for non-involutive implicative bilattices; finally, Section 5 concludes the paper.

2 Preliminaries

In what follows, we will introduce well known concepts of bilattices. See [6] for a gentle introduction to bilattices theory.

Definition 1. *A* pre-bilattice *is an algebra* $B = \langle B, \wedge, \vee, \sqcap, \sqcup\rangle$ *such that* $\langle B, \wedge, \vee\rangle$ *and* $\langle B, \sqcap, \sqcup\rangle$ *are both lattices.*

The lattice $\langle B, \wedge, \vee\rangle$ is called the truth lattice or t-lattice; its order is denoted by $\leq_t$ and is called the truth, $t-$order. The lattice $\langle B, \sqcap, \sqcup\rangle$ is called the knowledge lattice or k-lattice and its order $\leq_k$ the knowledge, $k-$order.

A pre-bilattice $B = \langle B, \wedge, \vee, \sqcap, \sqcup\rangle$ is called *interlaced* whenever each one of the four operations $\{\vee, \wedge, \sqcup, \sqcap\}$ is monotonic with respect to both orders $\leq_t$ and $\leq_k$.

Let $L_+ = \langle L_+, \wedge_+, \vee_+ \rangle$ or $L_= \langle L_-, \wedge_-, \vee_- \rangle$ be lattices with associated orders $\leq_+$ and $\leq_-$, respectively. The *product pre-bilattice* $\mathbf{L}_+ \odot \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup \rangle$ is defined as follows. For all $(a, b), (c, d) \in L_+ \times L_-$:

1. $(a, b) \wedge (c, d) = (a \wedge_+ c, b \vee_- d)$.
2. $(a, b) \vee (c, d) = (a \vee_+ c, b \wedge_- d)$.
3. $(a, b) \sqcap (c, d) = (a \wedge_+ c, b \wedge_- d)$.
4. $(a, b) \sqcup (c, d) = (a \vee_+ c, b \vee_- d)$.

The algebra $\mathbf{L}_+ \odot \mathbf{L}_-$ is always an interlaced pre-billatice, and, from the definition, it follows that:

$$(a, b) \leq_t (c, d) \text{ iff } a \leq_+ c \text{ and } d \leq_- b$$

$$(a, b) \leq_k (c, d) \text{ iff } a \leq_+ c \text{ and } b \leq_- d$$

That is, a member $(x, y) \in \mathbf{L}_+ \odot \mathbf{L}_-$ can be thought as encoding evidence about some assertion: evidence for, x, and evidence against, y. Then an increase in information (knowledge) amounts to saying that evidence in general goes up. An increase in truth implies that *evidence for* increases while *evidence against* decreases.

3 Non-involutive product bilattices

Let $\mathbf{L}_+ = \langle L_+, \wedge_+, \vee_+ \rangle$ and $\mathbf{L}_- = \langle L_-, \wedge_-, \vee_- \rangle$ be lattices and let $^-\colon L_+ \to L_-$ and $^+\colon L_- \to L_+$ be maps between them. We can construct the **non-involutive product bilattice**

$$\mathbf{L}_+ \bowtie \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup, \neg \rangle$$

as follows: $\langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup \rangle$ is the product pre-bilattice as defined above and

$$\neg \langle \alpha_+, \alpha_- \rangle := \langle (\alpha_-)^+, (\alpha_+)^- \rangle.$$

Definition 2. *A* non-involutive bilattice *is an interlaced pre-bilattice* $B = \langle B, \wedge, \vee, \sqcap, \sqcup, \neg \rangle$ *with the properties:*

$$\neg(x \wedge y) \equiv_+ \neg(x \sqcup y) \qquad \neg(x \wedge y) \equiv_- \neg(x \sqcap y)$$

where

$$\equiv_+ := \{\langle \alpha, \beta \rangle \in B \times B : \alpha \wedge \beta = \alpha \sqcup \beta\}$$

$$\equiv_- := \{\langle \alpha, \beta \rangle \in B \times B : \alpha \wedge \beta = \alpha \sqcap \beta\}.$$

Theorem 1. *Every non-involutive product bilattice* $\mathbf{L}_+ \bowtie \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup, \neg \rangle$ *is a non-involutive bilattice.*

Proof. Since $\mathbf{L}_+ \bowtie \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup, \neg \rangle$ is an interlaced pre-bilattice, the only thing we need to prove is that $\neg(x \wedge y) \equiv_+ \neg(x \sqcup y)$ and $\neg(x \wedge y) \equiv_- \neg(x \sqcap y)$.

Let $(a, b), (c, d) \in \mathbf{L}_+ \bowtie \mathbf{L}_-$. Then,

$$\neg((a, b) \wedge (c, d)) = ((b \vee d)^+, (a \wedge c)^-) \text{ and } \neg((a, b) \sqcup (c, d)) = ((b \vee d)^+, (a \vee c)^-)$$

Thus,

$$\neg((a,b)\wedge(c,d))\wedge\neg((a,b)\sqcup(c,d)) = ((b\vee d)^+\wedge(b\vee d)^+, (a\wedge c)^-\vee(a\vee c)^-) = ((b\vee d)^+, (a\wedge c)^-\vee(a\vee c)^-)$$

and

$$\neg((a,b)\wedge(c,d))\sqcup\neg((a,b)\sqcup(c,d)) = ((b\vee d)^+\vee(b\vee d)^+, (a\wedge c)^-\vee(a\vee c)^-) = ((b\vee d)^+, (a\wedge c)^-\vee(a\vee c)^-)$$

That is,

$$\neg((a,b)\wedge(c,d))\wedge\neg((a,b)\sqcup(c,d)) = \neg((a,b)\wedge(c,d))\sqcup\neg((a,b)\sqcup(c,d))$$

Hence

$$\neg((a,b)\wedge(c,d)) \equiv_+ \neg((a,b)\sqcup(c,d))$$

The proof of $\neg((a,b)\wedge(c,d)) \equiv_- \neg((a,b)\sqcup(c,d))$ is similar.

Therefore, $\mathbf{L}_+ \bowtie \mathbf{L}_-$ is a non-involutive bilattice. □

We will denote for $[a]_+$ and $[a]_-$ the equivalence class of a in $B/\equiv_+$ and $B/\equiv_+$, respectively.

The proof of the following lemma is given in [4], Proposition 3.8.1.

Lemma 1. *Let $B = \langle B, \wedge, \vee, \sqcap, \sqcup, \neg\rangle$ a non-involutive bilattice. Then, $B/\equiv_+$ and $B/\equiv_-$ are lattices with the operations:*

$$[a]_+ \sqcup [b]_+ = [a \sqcup b]_+$$

$$[a]_+ \sqcap [b]_+ = [a \sqcap b]_+$$

$$[a]_- \sqcup [b]_- = [a \sqcup b]_-$$

$$[a]_- \sqcap [b]_- = [a \sqcap b]_-$$

Hence we have the following result.

Lemma 2. *$B = \langle B, \wedge, \vee, \sqcap, \sqcup, \neg\rangle$ is a non-involutive bilattice iff $x \equiv_+ y \Rightarrow \neg x \equiv_- \neg y$ and $x \equiv_- y \Rightarrow \neg x \equiv_+ \neg y$.*

Proof. Let $B = \langle B, \wedge, \vee, \sqcap, \sqcup, \neg\rangle$ a non-involutive billatice and $\alpha \equiv_+ \beta$, i.e. $\alpha \wedge \beta = \alpha \sqcup \beta$. Then,

$$\begin{aligned}
\neg\alpha &= \neg(\alpha \sqcap (\alpha \sqcup \beta)) && \text{lattice identities}\\
&= \neg(\alpha \sqcap (\alpha \wedge \beta)) && \alpha \wedge \beta = \alpha \sqcup \beta\\
&\equiv_- \neg(\alpha \wedge \alpha \wedge \beta) && \neg(x \wedge y) \equiv_- \neg(x \sqcap y)\\
&= \neg(\beta \wedge \alpha \wedge \beta) && \text{lattice identities}\\
&\equiv_- \neg(\beta \sqcap (\alpha \wedge \beta)) && \neg(x \wedge y) \equiv_- \neg(x \sqcap y)\\
&= \neg(\beta \sqcap (\alpha \sqcup \beta)) && \alpha \wedge \beta = \alpha \sqcup \beta\\
&= \neg\beta && \text{lattice identities.}
\end{aligned}$$

We conclude that $\neg\alpha \equiv_- \neg\beta$ as required.

Similarly, using $\neg(x \wedge y) \equiv_+ \neg(x \sqcap y)$ we have $\alpha \equiv_+ \beta$ implies $\neg\alpha \equiv_- \neg\beta$. This shows that, for every non-involutive bilattice, we have $x \equiv_+ y \Rightarrow \neg x \equiv_- \neg y$ and $x \equiv_- y \Rightarrow \neg x \equiv_+ \neg y$.

The converse is easy, because the interlacing conditions imply that e.g. $x \wedge y \equiv_+ x \sqcap y$ and so by applying the $\alpha \equiv_+ \beta \Rightarrow \neg\alpha \equiv_- \neg\beta$ we obtain e.g. $\neg(x \wedge y) \equiv_- \neg(x \sqcap y)$. Similarly, for $\neg(x \wedge y) \equiv_+ \neg(x \sqcup y)$. □

Lemma 3. *Let* $\mathbf{L}_+ \bowtie \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup, \neg \rangle$ *a non-involutive product bilattice. Then,*

1. $[(\alpha_1, \alpha_2)]_+ = [(\beta_1, \beta_2)]_+$ *iff* $\alpha_1 = \beta_1$.
2. $[(\alpha_1, \alpha_2)]_- = [(\beta_1, \beta_2)]_-$ *iff* $\alpha_2 = \beta_2$.
3. $[(\alpha_1, \alpha_2)]_+ \leq [(\beta_1, \beta_2)]_+$ *iff* $\alpha_1 \leq \beta_1$.
4. $[(\alpha_1, \alpha_2)]_- \leq [(\beta_1, \beta_2)]_-$ *iff* $\alpha_2 \leq \beta_2$.

Proof. Let $[(\alpha_1, \alpha_2)]_+ = [(\beta_1, \beta_2)]_+$. Then, $(\alpha_1, \alpha_2) \wedge (\beta_1, \beta_2) = (\alpha_1, \alpha_2) \sqcup (\beta_1, \beta_2)$, that is, $(\alpha_1 \wedge \beta_1, \alpha_2 \vee \beta_2) = (\alpha_1 \vee \beta_1, \alpha_2 \vee \beta_2)$, thus, $\alpha_1 \wedge \beta_1 = \alpha_1 \vee \beta_1$. Therefore $\alpha_1 = \beta_1$. The conversely is easy to see, and 2 is similar to 1.

Now, let $[(\alpha_1, \alpha_2)]_+ \leq [(\beta_1, \beta_2)]_+$. Then, $[(\alpha_1, \alpha_2)]_+ \sqcap [(\beta_1, \beta_2)]_+ = [(\alpha_1, \alpha_2)]_+$, that is, $[(\alpha_1 \wedge \beta_1, \alpha_2 \sqcap \beta_2)]_+ = (\alpha_1, \alpha_2)]_+$, thus, $\alpha_1 \wedge \beta_1 = \alpha_1$, i.e, $\alpha_1 \leq \beta_1$. 4 can be proved similarly. □

Since $B/\equiv_+$ and $B/\equiv_-$ are lattices, we can define $(.)^+ : B/\equiv_- \longrightarrow B/\equiv_+$ and $(.)^- : B/\equiv_+ \longrightarrow B/\equiv_-$ by $([x]_-)^+ = [\neg x]_+$ and $([x]_+)^- = [\neg x]_-$.

Indeed, $(.)^+$ and $(.)^-$ are well-defined, because $x \equiv_+ y \Rightarrow \neg x \equiv_- \neg y$ and $x \equiv_- y \Rightarrow \neg x \equiv_+ \neg y$.

Then, we can see $B/\equiv_+ \bowtie B/\equiv_-$ for:

For all $([a], [b]), ([c], [d]) \in B/\equiv_+ \times B/\equiv_-$,

1. $([a]_+, [b]_-) \vee ([c]_+, [d]_-) = ([a]_+ \sqcup [c]_+, [b]_- \sqcap [d]_-)$.
2. $([a]_+, [b]_-) \wedge ([c]_+, [d]_-) = ([a]_+ \sqcap [c]_+, [b]_- \sqcup [d]_-)$.
3. $([a]_+, [b]_-) \sqcup ([c]_+, [d]_-) = ([a]_+ \sqcup [c]_+, [b]_- \sqcup [d]_-)$.
4. $([a]_+, [b]_-) \sqcap ([c]_+, [d]_-) = ([a]_+ \sqcap [c]_+, [b]_- \sqcap [d]_-)$.
5. $\neg([a]_+, [b]_-) = (([b]_-)^+, ([a]_+)^-)$.

Theorem 2. *Let* $B = \langle B, \wedge, \vee, \sqcap, \sqcup, \neg \rangle$ *a non-involutive bilattice. Then,* $\eta_B : B \longrightarrow B/\equiv_+ \bowtie B/\equiv_-$ *defined for* $\eta_B(x) = ([x]_+, [x]_-)$ *is an isomorphism.*

Proof. η_B is injective, surjective and preserves $\wedge, \vee, \sqcup, \sqcap$ was proved in [[4], Proposition 3.8.6]. It remains to show, $\eta(\neg x) = \neg\eta(x)$. We have $\eta(\neg x) = ([\neg x]_+, [\neg x]_-)$ and $\neg\eta(x) = \neg([x]_+, [x]_-) = (([x]_-)^+, ([x]_+)^-)$. So $\eta(\neg x) = \neg\eta(x)$, since $= (([x]_-)^+, ([x]_+)^-) = ([\neg x]_+, [\neg x]_-)$. □

We are going to see that η_B is in fact the unit of categorical equivalence between two naturally associated categories.

The category NIB has as objects non-involutive bilattices and as morphisms algebraic non-involutive bilattice homomorphisms. On the other side of our equivalence, the category $NIPB$ has as objects 4-tuples $L = (L_+, L_-, ()^+, ()^-)$ with L_+ and L_- lattices and $()^+$ and $()^-$ maps between then. A morphism between $NIPB$-objects are $h : (L_{1+}, L_{1-}, ()_1^+, ()_2^-) \rightarrow$

$(L_{2+}, L_{2-}, ()_{2^+}, ()_{2^-})$ such that h_+ and h_- are lattices homomorphisms and $h_+ \circ ()_1^+ = ()_2^+ \circ h_-$ and $h_- \circ ()_1^- = ()_2^- \circ h_+$.

We proceed to define functors $T : NIB \longrightarrow NIPB$ and $N : NIPB \longrightarrow NIB$ that will allow us to prove the equivalence between the two categories.

Given a non-involutive bilattice B, we let $T(B) := (B/\equiv_+, B/\equiv_-, ()^+, ()^-)$ with $([x]_-)^+ = [\neg x]_+$ and $([x]_+)^- = [\neg x]_-$ for all x. If $f : B_1 \longrightarrow B_2$ is a NIB-morphism, we define $T(f) : T(B_1) \longrightarrow T(B_2)$ as $T(f)([\alpha]_+, [\beta]_-) = ([f(\alpha)]_+, [f(\beta)]_-)$.

Lemma 4. *T is a covariant functor.*

Proof. Indeed $T(B_1)$ and $T(B_2)$ are NIPB-objects since B_1 and B_2 are non-involutive bilattices. It is to see that $T(f)$ is well-defined, $T(f)$ is a $NIPB$-morphism, $f(1_B) = 1_{T(B)}$ and $T(f \circ g)([\alpha]_+, [\beta]_-) = ([(f \circ g)(\alpha)]_+, [(f \circ g)(\beta)]_-) = ([f(g(\alpha))]_+, [f(g(\beta))]_-) = (T(f) \circ T(g))([\alpha]_+, [\beta]_-)$ for all α and β, that is $T(f \circ g) = T(f) \circ T(g)$.

Therefore T is a covariant functor. □

Conversely, for $L = (L_+, L_-, ()^+, ()^-)$ a NIPB-object, we let $N(L) = L_+ \bowtie L_-$. We know by Theorem 3 that $L_+ \bowtie L_-$ is a non-involutive bilattice. For a morphism $h : L_1 \to L_2$ between NIPB-objects, we define the map $N(h) : N(L_1) \to N(L_2)$, for all $a, b \in L_1$, as $N(h)(a, b) := (h_+(a), h_-(b))$. Is easy to see that N is a covariant functor.

Therefore, by Theorem 7, for any B non-involutive bilattice, the map $\eta_B : B \to N(T(B))$ is an isomorphism.

Theorem 3. *For any NIPB-object L, the map $\varepsilon_L : L \to T(N(L))$ defined by $\varepsilon_L(a, b) = ([(a, b)]_+, [(a, b)]_-)$ is an isomorphism.*

Proof. Let $(a_1, b_1), (a_2, b_2) \in L$.

If $(a_1, b_1) = (a_2, b_2)$ then $\varepsilon_L(a_1, b_1) = ([(a_1, b_1)]_+, [(a_1, b_1)]_-) = ([(a_2, b_2)]_+, [(a_2, b_2)]_-) = \varepsilon_L(a_2, b_2)$, that is ε_L is well-defined.

If $\varepsilon_L(a_1, b_1) = \varepsilon_L(a_2, b_2)$, we have $([(a_1, b_1)]_+, [(a_1, b_1)]_-) = ([(a_2, b_2)]_+, [(a_2, b_2)]_-)$ then $a_1 = a_2$ since $[(a_1, b_1)]_+ = [(a_2, b_2)]_+$ and $b_1 = b_2$ since $[(a_1, b_1)]_- = [(a_2, b_2)]_-$, that is $(a_1, b_1) = (a_2, b_2)$. In other words, ε_L is injective.

Let $([(a_1, b_1)]_+, [(a_2, b_2)]_-) \in T(N(L))$. Since $([(a_1, b_1)]_+, [(a_2, b_2)]_-) = ([(a_1, b_2)]_+, [(a_2, b_2)]_-) = ([(a_1, b_2)]_+, [(a_1, b_2)]_-) = \varepsilon_L(a_1, b_2)$, ε_L is surjective.

Lastly, ε_L is a NIPB-morphism. Therefore, ε_L is an isomorphism. □

Theorem 4. *Let $f : B_1 \to B_2$ be an NIB-morphism. Then $N(T(f)) \circ \eta_{B_1} = \eta_{B_2} \circ f$.*

Proof. Let $x \in B_1$. Then $(N(T(f)) \circ \eta_{B_1})(x) = N(T(f))([x]_+, [x]_-) = (T(f)_+([x]_+), T(f)_-([x]_-)) = ([f(x)]_+, [f(x)]_-) = \eta_{B_2}(f(x)) = (\eta_{B_2} \circ f)(x)$.

Therefore, $N(T(f)) \circ \eta_{B_1} = \eta_{B_2} \circ f$. □

Theorem 5. *Let $h : L_1 \to L_2$ be an NIPB-morphism. Then $T(N(h)) \circ \varepsilon_{L_1} = \varepsilon_{L_2} \circ h$.*

Proof. Let $(x, y) \in L_1$. Then $(T(N(h)) \circ \varepsilon_{L_1})(x, y) = (T(N(h))([(x, y)]_+, [(x, y)]_-) = ([N(h)(x, y)]_+, [N(h)(x, y)]_-) = ([(h_+(x), h_-(y))]_+, [(h_+(x), h_-(y))]_-) = ([h(x, y)]_+, [h(x, y)]_-) = \varepsilon_{L_2}(h(x, y)) = (\varepsilon_{L_2} \circ h)(x, y)$.

Therefore, $T(N(h)) \circ \varepsilon_{L_1} = \varepsilon_{L_2} \circ h$. □

Theorem 6. *Functors $T : NIB \to NIPB$ and $N : NIPB \to NIB$ stablish a natural equivalence between the category NIB and NIPB.*

$$\begin{array}{ccc} B_1 & \xrightarrow{\eta_{B_1}} & N(T(B_1)) \\ \downarrow f & & \downarrow N(T(f)) \\ B_2 & \xrightarrow{\eta_{B_2}} & N(T(B_2)) \end{array} \qquad \begin{array}{ccc} L_1 & \xrightarrow{\epsilon_{L_1}} & T(N(L_1)) \\ \downarrow h & & \downarrow T(N(h)) \\ L_2 & \xrightarrow{\epsilon_{L_2}} & T(N(L_2)) \end{array}$$

4 Non-involutive implicative product bilattices

A Brouwerian lattice is a lattice L with the property $c \wedge a \leq b$ iff $c \leq a \to b$ for all $a, b, c \in L$.

Suppose $\mathbf{L}_+ = \langle L_+, \wedge_+, \vee_+, \to_+ \rangle$ and $\mathbf{L}_- = \langle L_-, \wedge_-, \vee_-, \to_- \rangle$ are Brouwerian lattices and $^-\colon L_+ \to L_-$ and $^+\colon L_- \to L_+$ are maps between them. Then we can construct the **non-involutive implicative product bilattice**

$$\mathbf{L}_+ \bowtie \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup, \supset, \subset \neg \rangle$$

as follows: $\langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup \rangle$ is the above-defined non-involutive product bilattice and

$$\langle \alpha_+, \alpha_- \rangle \supset \langle \beta_+, \beta_- \rangle := \langle \alpha_+ \to_+ \beta_+,\ (\alpha_+)^- \wedge_- \beta_- \rangle$$

$$\langle \alpha_+, \alpha_- \rangle \subset \langle \beta_+, \beta_- \rangle := \langle \alpha_+ \wedge_+ (\beta_-)^+,\ \beta_- \to_- \alpha_- \rangle$$

Definition 3. *A* non-involutive implicative bilattice $B = \langle B, \wedge, \vee, \sqcap, \sqcup, \neg, \to, \leftarrow \rangle$ *is a non-involutive bilattice such that* $\langle B/\equiv_+, \wedge, \vee, \to \rangle$ *and* $\langle B/\equiv_-, \sqcap, \sqcup, \leftarrow \rangle$ *are Browerian lattices, and:*

$$x \leftarrow y \equiv_+ x \sqcap \neg y$$

$$x \to y \equiv_- \neg x \sqcap y$$

.

Theorem 7. *Every* $\mathbf{L}_+ \bowtie \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup, \neg, \supset, \subset \rangle$ *non-involutive implicative product bilattice is a non-involutive implicative bilattice.*

Proof. It is easy to see that operations are compatible.

We need to show that $\mathbf{L}_+ \bowtie \mathbf{L}_-/\equiv_+$ and $\mathbf{L}_+ \bowtie \mathbf{L}_-/\equiv_-$ are Browerian lattices. Indeed, let $[(\alpha_1, \alpha_2)], [(\beta_1, \beta_2)], [(\gamma_1, \gamma_2)] \in \mathbf{L}_+ \bowtie \mathbf{L}_-/\equiv_+$. If $[(\alpha_1, \alpha_2)] \sqcap [(\beta_1, \beta_2)] \leq [(\gamma_1, \gamma_2)]$, we have $[(\alpha_1, \alpha_2) \sqcap (\beta_1, \beta_2)] \leq [(\gamma_1, \gamma_2)]$, thus, $[(\alpha_1 \wedge_+ \beta_1, \alpha_2 \wedge_- \beta_2)] \leq [(\gamma_1, \gamma_2)]$, that is, $\alpha_1 \wedge_+ \beta_1 \leq \gamma_1$, then, $\alpha_1 \leq \beta_1 \to_+ \gamma_1$ since $\mathbf{L}_+$ is a Browerian lattice. Therefore, $[(\alpha_1, \alpha_2)] \leq [(\beta_1 \to_+ \gamma_1, (\beta_1)^- \wedge_- \gamma_2)]$, thus, $[(\alpha_1, \alpha_2)] \leq [(\beta_1, \beta_2) \supset (\gamma_1, \gamma_2)]$, that is, $[(\alpha_1, \alpha_2)] \leq [(\beta_1, \beta_2)] \supset [(\gamma_1, \gamma_2)]$, i.e., $\mathbf{L}_+ \bowtie \mathbf{L}_-/\equiv_+$ is a Browerian lattice. Similarly, using the $\subset$ definition, we can prove that $\mathbf{L}_+ \bowtie \mathbf{L}_-/\equiv_-$ is a Browerian lattice.

Now, let $(\alpha_1, \alpha_2), (\beta_1, \beta_2) \in \mathbf{L}_+ \bowtie \mathbf{L}_-$. Then, $(\alpha_1, \alpha_2) \subset (\beta_1, \beta_2) = (\alpha_1 \wedge_+ (\beta_2)^+, \beta_2 \to_- \alpha_2)$ and $(\alpha_1, \alpha_2) \sqcap \neg(\beta_1, \beta_2) = (\alpha_1, \alpha_2) \sqcap ((\beta_2)^+, (\beta_1)^-) = (\alpha_1 \wedge_+ (\beta_2)^+, \alpha_2 \wedge_- (\beta_1)^-)$, that is, $(\alpha_1, \alpha_2) \subset (\beta_1, \beta_2) \equiv_+ (\alpha_1, \alpha_2) \sqcap \neg(\beta_1, \beta_2)$. Also $(\alpha_1, \alpha_2) \supset (\beta_1, \beta_2) = [\alpha_1 \to_+ \beta_1, (\alpha_1)^- \wedge_- \beta_2]$ and $\neg(\alpha_1, \alpha_2) \sqcap (\beta_1, \beta_2) = ((\alpha_2)^+ \wedge_+ \beta_1, (\alpha_1)^- \wedge_- \beta_2)$, that is, $(\alpha_1, \alpha_2) \supset (\beta_1, \beta_2) \equiv_- \neg(\alpha_1, \alpha_2) \sqcap (\beta_1, \beta_2)$.

Therefore, $\mathbf{L}_+ \bowtie \mathbf{L}_- = \langle L_+ \times L_-, \wedge, \vee, \sqcap, \sqcup, \neg, \supset, \subset \rangle$ is a non-involutive implicative billatice. □

Now we can construct $B/\equiv_+ \bowtie B/\equiv_-$ with the operations:

For all $([a],[b]),([c],[d]) \in B/\equiv_+ \times B/\equiv_-$,

1. $([a]_+,[b]_-) \vee ([c]_+,[d]_-) = ([a]_+ \sqcup [c]_+, [b]_- \sqcap [d]_-)$.
2. $([a]_+,[b]_-) \wedge ([c]_+,[d]_-) = ([a]_+ \sqcap [c]_+, [b]_- \sqcup [d]_-)$.
3. $([a]_+,[b]_-) \sqcup ([c]_+,[d]_-) = ([a]_+ \sqcup [c]_+, [b]_- \sqcup [d]_-)$.
4. $([a]_+,[b]_-) \sqcap ([c]_+,[d]_-) = ([a]_+ \sqcap [c]_+, [b]_- \sqcap [d]_-)$.
5. $\neg([a]_+,[b]_-) = (([b]_-)^+, ([a]_+)^-)$.
6. $([a]_+,[b]_-) \supset ([c]_+,[d]_-) = ([a]_+ \to_+ [c]_+, ([a]_+)^- \sqcap [d]_-)$.
7. $([a]_+,[b]_-) \subset ([c]_+,[d]_-) = ([a]_+ \sqcap ([d]_-)^+, [b]_- \to_- [d]_-)$.

Theorem 8. *Let $B = \langle B, \wedge, \vee, \sqcap, \sqcup, \neg, \to, \leftarrow\rangle$ a non-involutive implicative bilattice. Then, $\eta_B : B \longrightarrow B/\equiv_+ \bowtie B/\equiv_-$ defined for $\eta_B(x) = ([x]_+,[x]_-)$ is an isomorphism.*

Proof. We already know that η_B is injective, surjective and preserves $\wedge, \vee, \sqcup, \sqcap, \neg$. It remains to show $\eta_B(x \to y) = \eta_B(x) \supset \eta_B(y)$ and $\eta_B(x \leftarrow y) = \eta_B(x) \subset \eta_B(y)$.

Indeed, $\eta_B(x \to y) = ([x \to y]_+, [x \to y]_-) = ([x]_+ \to_+ [y]_+, [x \to y]_-)$ and $\eta_B(x) \supset \eta_B(y) = ([x]_+,[x]_-) \supset ([y]_+,[y]_-) = ([x]_+ \to_+ [y]_+, ([x]_+)^- \sqcap [y]_-) = ([x]_+ \to_+ [y]_+, ([\neg x]_- \sqcap [y]_-) = ([x]_+ \to_+ [y]_+, [\neg x \sqcap y]_-)$. Then, $\eta_B(x \to y) = \eta_B(x) \supset \eta_B(y)$.

Also, $\eta_B(x \leftarrow y) = ([x \leftarrow y]_+, [x \leftarrow y]_-) = ([x \leftarrow y]_+, [x]_- \to_- [y]_-)$ and $\eta_B(x) \subset \eta_B(y) = ([x]_+,[x]_-) \subset ([y]_+,[y]_-) = ([x \sqcap \neg y]_+, [x]_- \to_- [y]_-)$. Then, $\eta_B(x \leftarrow y) = \eta_B(x) \subset \eta_B(y)$.

Therefore, η_B is an isomorphism. □

5 Conclusion and future work

We provided equational presentations for the class of all non-involutive bilattices and the subclasses corresponding to bilattices with an involutive negation, bilattices with implication etc. For each of these we proved a representation theorem and a categorical equivalence for non-involutive bilattices, that allows us to view any algebra and their morphism in the class as a bilattice product of two lattices and their morphisms.

For future work we pretend to do a categorical equivalence for non-involutive implicative bilattices and non-involutive implicative product bilattices and also a topological duality for both. And characterize the congruences of non-involutive bilattices and non-involutive implicative bilattices.

References

1. Arieli, Ofer, and Arnon Avron, 'The value of the four values', *Artificial Intelligence*, 102 (1998), 1, 97–141.
2. Belnap Jr, Nuel D, 'A useful four-valued logic', in *Modern uses of multiple-valued logic*, Springer, 1977, pp. 5–37.
3. Bou, Félix, Ramon Jansana, and Umberto Rivieccio, 'Varieties of interlaced bilattices', *Algebra universalis*, 66 (2011), 1-2, 115–141.
4. Bou, Félix, and Umberto Rivieccio, 'The logic of distributive bilattices', *Logic Journal of IGPL*, 19 (2011), 1, 183–216.

5. Cabrer, Leonardo Manuel, Andrew P.K. Craig, and Hilary A Priestley, 'Product representation for default bilattices: an application of natural duality theory', *Journal of Pure and Applied Algebra*, 219 (2015), 7, 2962–2988.
6. Fitting, Melvin, 'Bilattices in logic programming', in *Multiple-Valued Logic, 1990., Proceedings of the Twentieth International Symposium on*, IEEE, 1990, pp. 238–246.
7. Ginsberg, Matthew L, 'Multivalued logics: A uniform approach to reasoning in artificial intelligence', *Computational intelligence*, 4 (1988), 3, 265–316.
8. Jakl, Tomáš, Achim Jung, and Aleš Pultr, 'Bitopology and four-valued logic', in Lars Birkedal, and Michael Mislove, (eds.), *32nd Conference on Mathematical Foundations of Programming Semantics*, 2016.
9. Jung, Achim, and M Andrew Moshier, 'On the bitopological nature of stone duality', *School of Computer Science Research Reports-University of Birmingham CSR*, 13 (2006).
10. Jung, Achim, and Umberto Rivieccio, 'Priestley duality for bilattices', *Studia Logica*, 100 (2012), 1-2, 223–252.
11. Mobasher, B, D Pigozzi, G Slutzki, and G Voutsadakis, 'A duality theory for bilattices', *Algebra universalis*, 43 (2000), 2-3, 109–125.

Comparação de codificações para solução de puzzles Sudoku via algoritmo DPLL

Savio Lopes Rabelo, Helio Henrique Barbosa Rocha, e Thiago Alves Rocha

Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE), Eixo Tecnológico de Computação, Campus Maracanaú – CE – Brazil
saviorabelo.ti@gmail.com, hique.rocha@gmail.com, thiago.alves@ifce.edu.br

Resumo Na presente investigação, exemplos de puzzles Sudoku serão codificados como problemas SAT. O propósito deste trabalho é avaliar o desempenho do algoritmo DPLL para resolver puzzles Sudoku considerando formas de codificação para sua solução: minimal e estendida. O algoritmo e as codificações foram implementados na linguagem de programação Python. Exemplos de entrada de jogos aleatórios foram considerados. O tempo para solução do jogo pelo algoritmo pode depender, além da codificação empregada, da configuração do mesmo.

Palavras-chave: lógica computacional, codificação SAT, algoritmo DPLL

1 Introdução

Sudoku é um puzzle combinatório baseado no posicionamento lógico de números. Ainda que as suas regras possam ser consideradas simples, a sua resolução pode se tornar um desafio intelectual. Na sua versão padrão, o jogo tem por objetivo a colocação dos números de 1 a 9 em cada uma das células vazias numa grade de dimensão 9x9 (que contem 81 células), constituída por sub-grades de dimensão 3x3 denominadas regiões [10]. Há, portanto, 9 regiões e cada região contém 9 células. Inicialmente o quebra-cabeça está parcialmente preenchido, contendo números inseridos em algumas células, de maneira a permitir uma indução ou dedução dos números em células que estejam vazias. Em cada coluna, linha e região, os números de 1 a 9 aparecem uma única vez.

Pela simplicidade de suas regras e também por ser membro de uma classe de problemas de satisfação de restrições, o Sudoku tem sido objeto de muitos estudos, em particular no que diz respeito às suas propriedades matemáticas e algorítmicas, como por exemplo enumeração de possíveis grades de jogo [6,7] e NP-completude da sua versão generalizada [18]. Além disso, vários métodos foram utilizados para resolver o Sudoku: Formulações em problemas de satisfação de restrições [17,15], métodos de busca [8], algoritmos genéticos [12] e o método *particle swarm optimization* [14].

Uma maneira de resolver tais puzzles combinatórios consiste na sua modelagem como um problema de satisfazibilidade (SAT) da lógica proposicional. O problema SAT envolve a busca por uma atribuição de valores verdade que

torna verdadeira uma fórmula da lógica proposicional [16]. O SAT foi o primeiro problema identificado como NP-Completo[1], sendo ainda hoje um dos mais estudados dessa classe. Destacam-se na literatura recente, trabalhos que tratam de problemas SAT com o uso de hipergrafos. Foram evidenciadas em [2] novas perspectivas sobre a representação de problemas SAT com base em hipergrafos direcionados, além de um novo algoritmo tipo DPLL. É notório o mérito no estudo destes problemas na medida em que resolvedores bem elaborados associados a codificações apropriadas favorecem a resolução de muitas instâncias úteis em diversas áreas do conhecimento. O algoritmo DPLL frequentemente serve como ponto de partida no desenvolvimento de métodos capazes de resolver de maneira eficiente instâncias do SAT. Será empregado o algoritmo DPLL[2] como descrito em [16] e com escolha de literais feita de forma aleatória.

O principal objetivo deste trabalho é estudar a codificação de puzzles Sudoku com fórmulas da lógica proposicional. É pretendido também comparar o desempenho do algoritmo DPLL com codificações diferentes e instâncias distintas do problema. Neste sentido, objetiva-se buscar a codificação mais eficiente para o caso geral do problema: dada uma situação inicial observar qual codificação apresenta melhor comportamento. Ainda, visa-se avaliar, considerando casos específicos, a importância da codificação para resolver de forma eficiente instâncias do problema. Assim sendo, a elucidação computacional de jogos Sudoku é desenvolvida considerando duas etapas: sua codificação em forma normal conjuntiva (FNC) com posterior resolução via algoritmo DPLL. São introduzidas duas codificações diretas para o Sudoku [11]: codificação minimal e codificação estendida. A codificação minimal é suficiente para caracterizar o jogo, enquanto a codificação estendida acrescenta cláusulas excedentes para a codificação minimal.

2 Codificações SAT para o problema Sudoku

A codificação minimal assegura que há pelo menos um número (entre 1 e 9) em cada célula, e que cada número aparece no máximo uma vez em cada linha, em cada coluna e em cada sub-grade 3x3. Considerando que os índices x,y e z da variável booleana p_{xyz} representam linha, coluna e número, respectivamente, formalmente temos [11]:

"Há, pelo menos, um número em cada célula":

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigvee_{z=1}^{9} p_{xyz}, \tag{1}$$

[1] Por ser NP-completo, não se conhece algoritmo com tempo melhor (no pior caso) que o exponencial.

[2] O DPLL é reconhecidamente capaz de decidir corretamente se um conjunto de cláusulas é satisfazível ou não e adequado por aceitar novas heurísticas para escolha de literais quando implementado na resolução de problemas SAT

"Cada número aparece, no máximo, uma vez em cada linha":

$$\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{x=1}^{8}\bigwedge_{i=x+1}^{9}(\neg p_{xyz} \vee \neg p_{iyz}), \tag{2}$$

"Cada número aparece, no máximo, uma vez em cada coluna":

$$\bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{y=1}^{8}\bigwedge_{i=y+1}^{9}(\neg p_{xyz} \vee \neg p_{xiz}), \tag{3}$$

"Cada número aparece, no máximo, uma vez em cada sub-grade (3x3)":

$$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=y+1}^{3}(\neg p_{(3i+x)(3j+y)z} \vee \neg p_{(3i+x)(3j+k)z}), \tag{4}$$

$$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=x+1}^{3}\bigwedge_{l=1}^{3}(\neg p_{(3i+x)(3j+y)z} \vee \neg p_{(3i+k)(3j+l)z}). \tag{5}$$

Na codificação minimal, a fórmula resultante terá 8829 cláusulas, sem contar as cláusulas unitárias representando as células pré-preenchidas. Destas cláusulas, 81 cláusulas têm tamanho nove e as 8748 restantes possuem tamanho dois.

A codificação estendida assegura que cada entrada na grade possui exatamente um número, e o mesmo para cada linha, cada coluna e cada sub-grade 3x3. A codificação estendida inclui todas as cláusulas da codificação minimal, bem como as seguintes restrições:

"Há, no máximo, um número em cada entrada":

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{8}\bigwedge_{i=z+1}^{9}(\neg p_{xyz} \vee \neg p_{xyi}), \tag{6}$$

"Cada número aparece pelo menos uma vez em cada linha":

$$\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{9}\bigvee_{x=1}^{9} p_{xyz}, \tag{7}$$

"Cada número aparece pelo menos uma vez em cada coluna":

$$\bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigvee_{y=1}^{9} p_{xyz}, \tag{8}$$

"Cada número aparece pelo menos uma vez em cada sub-grade (3 x 3)":

$$\bigvee_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3} p_{(3i+x)(3j+y)z}. \tag{9}$$

Na codificação estendida, a fórmula resultante possuirá 11988 cláusulas, descontadas as cláusulas unitárias representando as entradas pré-alocadas (contendo números). Destas cláusulas, 324 têm tamanho nove e as 11664 restantes possuem tamanho dois.

2.1 O algoritmo DPLL

O algoritmo DPLL (ou método Davis–Putnam–Logemann–Loveland) [3,4] vem sendo implementado com os mais diversos métodos ou processos para soluções de problemas [16]. Basicamente, uma valoração para uma fórmula fornecida (na forma de um conjunto de cláusulas) deve ser construída. No início, todos os átomos da fórmula apresentam um valor desconhecido para sua valoração. A cada iteração do algoritmo (descrito no Algoritmo 1 [16]), um literal é escolhido, ficando determinada sua valoração como sendo verdadeira. Caso esse literal seja negativo, a valoração da atômica do literal é falsa. Uma vez valorada, a fórmula é simplificada (Algoritmo 2 [16]) pela eliminação das cláusulas que contêm o literal e pela eliminação da negação do literal das cláusulas restantes. Se essa valoração satisfizer todas as cláusulas, tem-se uma valoração que satisfaz a fórmula inicial. Se alguma cláusula for falsificada, altera-se a escolha da valoração para falso. Se nenhuma cláusula for falsificada, nem todas as cláusulas foram satisfeitas, procede-se à próxima escolha de literal. O processo para quando uma valoração for encontrada (fórmula satisfazível) ou quando não há mais átomos para serem testados (fórmula insatisfazível). Para uma instância SAT, um algoritmo completo é aquele que acha uma solução (ou prova que tal solução não existe). O método DPLL apresentado abaixo é chamado de SAT-completo, ou seja, ele sempre é capaz de decidir corretamente se um conjunto de cláusulas é ou não é satisfazível.

Algoritmo 1: Algoritmo DPLL(F).

Entrada: Uma fórmula F no formato CNF.
Saída: verdadeiro, se F é satisfazível ou falso, caso contrário.

```
1  Fazer v(p) = * para todo átomo p;
2  F'= Simplifica(F);
3  se F' = ∅ então
4  |  retorna verdadeiro;
5  senão se F' contém uma cláusula vazia (falsa) então
6  |  retorna falso;
7  fim
8  Escolha um literal L com v(L) = *;
9  se DPPL(F' ∪ L) = verdadeiro então
10 |  retorna verdadeiro;
11 senão se DPPL(F' ∪ ¬L) = verdadeiro então
12 |  retorna verdadeiro;
13 senão
14 |  retorna falso;
15 fim
```

Algoritmo 2: Algoritmo Simplifica(F).

Entrada: Uma fórmula F no formato CNF.
Saída: Uma fórmula na forma clausal equivalente a F porém mais simples.

```
1 enquanto F possui alguma cláusula unitária L faça
2 |   Apaga de F todas cláusulas que contém L;
3 |   Apaga ¬L das cláusulas restantes;
4 fim
5 retorna F
```

3 Metodologia

Inicialmente, uma amostra de jogo escolhida ao acaso serve como entrada no programa. Neste momento, apenas a codificação minimal é considerada. Uma vez que o jogo tenha sido resolvido[3], cada célula contendo um número é sequencialmente eliminada. A cada eliminação de célula (cumulativamente) o jogo é novamente resolvido.

Devemos observar um ponto[4] a partir do qual a solução torna-se dispendiosa (mais lenta). Neste ponto deve se guardar a quantidade de células preenchidas. Em seguida, outros exemplos de jogos contendo aproximadamente a mesma quantidade de células preenchidas são analisados em relação ao tempo de solução. Nesta etapa os resultados obtidos pelas codificações minimal e estendida serão contrastados quanto ao seu impacto na solução.

Ao longo deste trabalho empregamos o algoritmo DPLL[5] que trata fórmulas no formato clausal (chamada de Forma Normal Conjuntiva). Esta restrição visa facilitar a implementação de resolvedores sem perda de generalidade[6]. O problema SAT consiste em determinar se uma fórmula na FNC é satisfazível ou não.

Os arquivos com instâncias de jogos aleatórias foram obtidos on-line[7], sendo posteriormente convertidos no formato padrão CNF. Os algoritmos 1 e 2 supramencionados [16] foram implementados na linguagem de programação Python versão 2.7.11, bem como as codificações SAT e as entradas de problemas Sudoku. Para efeito de reprodutibilidade, os requisitos mínimos de hardware são: Processador Intel Core i3, 250 MB de espaço livre em disco e 4 GB de memoria RAM. Os requisitos mínimos de software incluem sistema operacional Windows 10.

[3] As 81 células preenchidas com números conforme as regras do Sudoku.
[4] Tendo em vista a quantidade de células que restam na grade.
[5] O literal é escolhido aleatoriamente.
[6] É demonstrável que qualquer fórmula da lógica proposicional clássica tem uma fórmula equivalente no formato clausal.
[7] `http://www.sudoku.name/`

4 Resultados e Discussões

Tendo em vista os resultado obtidos quando da solução de um exemplo e jogo, as células são eliminadas uma a uma em sequência. Após esta eliminação o jogo é novamente resolvido, e, desta vez, o tempo para a sua resolução é registrado.

O resultado deste processo, a partir do ponto em que a grade do jogo encontrava-se preenchida com 62 células pré-preenchidas, está ilustrado na Fig. 1. Em cada eliminação foram realizadas 5 computações de tempo. Aparentemente, considerando apenas a codificação minimal, uma grande quantidade de células pré-preenchidas implica em uma menor variabilidade e tempo de solução. A partir do momento em que a grade passa a conter 55 números, o tempo para solucionar o jogo começa a sofrer aumento considerável acompanhado de uma maior variabilidade. Para a instância considerada, com 52 restrições não se fez possível resolver o jogo em menos de 11 minutos.

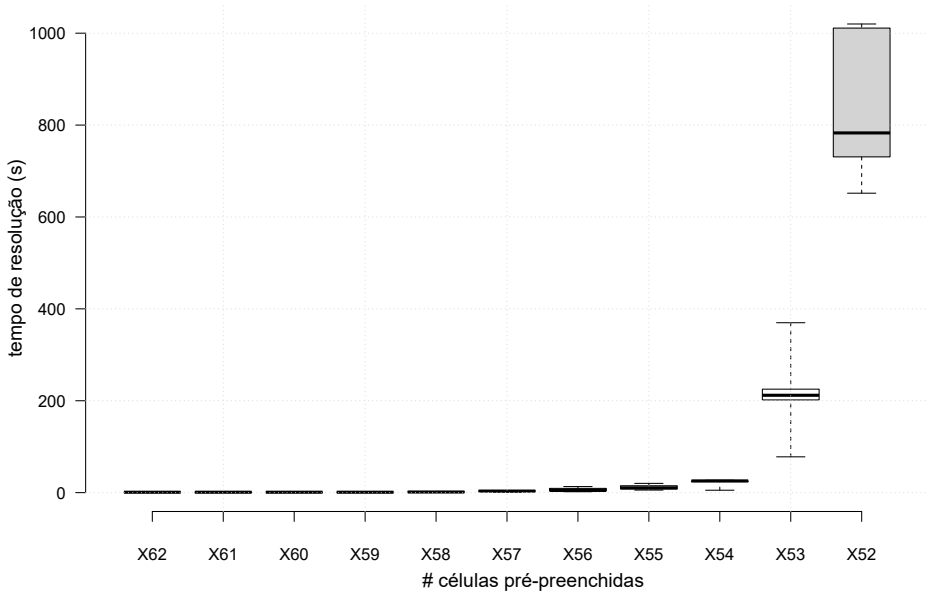


Figura 1: Tempo de resolução do jogo, usando codificação minimal (Eqs.1-5), em termos do número de células pré-preenchidas.

Foram tomadas aleatoriamente outras três instâncias de jogo diferentes quanto à quantidade de células pré-preenchidas. Estas amostras foram computadas considerando cinco situações, sendo a primeira apenas com codificação minimal e as posteriores contendo, além da codificação minimal, as codificações estendidas, acrescentadas acumulativamente uma por vez. Com isto, é pretendido analisar

o possível impacto da redundância no desempenho associado ao programa. Os resultados desta etapa encontram-se condensados na Tab. 1.

Tabela 1: Tempo médio (s) para resolução dos jogos.

	restrições				
# **células pré-preenchidas**	Eqs.(1)-(5)	Eqs.(1)-(6)	Eqs.(1)-(7)	Eqs.(1)-(8)	Eqs.(1)-(9)
54	4,062	6,422	6,360	6,547	6,563
53	3,703	6,297	6,359	6,359	6,516
45	3,688	6,247	6,203	6,344	6,344

Na Tab. 1, temos a quantidade de células pré-preenchidas em cada uma das três amostras analisadas e as restrições observadas, em que se considerou desde a codificação minimal (Eqs.1-5) até o acréscimo de todas as codificações estendidas (Eqs.1-9). Cada computação foi repetida sete vezes, sendo apresentado o valor médio resultante. Há duas observações sobre os resultados obtidos: o tempo de solução do jogo aumenta à medida que são incluídas as codificações estendidas, e a solução para um jogo com menos células pré-preenchidas é obtida num tempo menor. Além disto, para uma mesma amostra, o acréscimo das codificações estendidas não apresenta alterações significativas de tempo. Tal constatação aparentemente contrasta com as observações iniciais: quanto menos células pré-preenchidas, mais tempo se levaria para solucionar um problema. É sugerido que a quantidade de células pré-preenchidas, apesar de claramente importante, tem dominância possivelmente limitada sobre o resultado do experimento.

Continuando, insistiu-se na redução da quantidade de células pré-preenchidas alimentadas. Foi selecionado um jogo com apenas 22 células pré-preenchidas. Neste caso, a solução envolvendo unicamente a codificação minimal (Eqs.1-5) não se mostrou suficiente para obtenção de uma resposta no tempo estipulado de 20 minutos. No entanto, envolvendo a codificação estendida completa (Eqs.1-9), o mesmo jogo foi resolvido num tempo médio de 6.3 segundos.

5 Conclusão e Trabalhos Futuros

Neste trabalho estudamos o impacto da codificação em relação ao tempo de resposta de problemas SAT na forma de amostras de puzzles Sudoku com emprego do algoritmo DPLL para sua resolução. Embora não tenha sido verificado neste trabalho, é possível que, em geral, um jogo possua mais de uma solução. A unicidade da resolução de instâncias Sodoku é estudada em [13], sendo tal requisito implementado em [9], conforme descrito em [10]. A princípio, a quantidade parece influenciar no tempo de reposta obtido. Em conformidade com os experimentos realizados, jogos com menor quantidade de células preenchidas podem ser resolvidos em menor tempo que jogos com maior quantidade de células pré-preenchidas. Além disso, uma redução ainda maior da quantidade de células

pré-preenchidas supostamente evidencia a importância da redundância da codificação estendida no contexto da resolução do problema. Portanto, a configuração dos jogos parece exercer forte influência sobre o tempo em que os mesmos são elucidados. Isso foi analisado em outros trabalhos, como [5] e [19].

Para investigações posteriores, pretende-se fazer uma comparação com variações de heurísticas de escolha de literais no DPLL [16] e comparações com outras abordagens, a exemplo de algoritmos genéticos [12], *particle swarm optimization* [14] e *simulated annealing* [1].

Referências

1. Chi, E. C., Lange, K. Techniques for Solving Sudoku Puzzles. CoRR, abs/1203.2295, 2012.
2. Croitoru, C.; Croitoru, M. Combinatorial Results on Directed Hypergraphs for the SAT Problem. Graph Structures for Knowledge Representation and Reasoning: 4th International Workshop, GKR 2015, Buenos Aires, Argentina, July 25, 2015, Revised Selected Papers. Cham: Springer International Publishing, 2015. p. 72–88.
3. Davis, M.; Putnam, H. A Computing Procedure for Quantification Theory. J. ACM, v. 7, n. 3, p. 201–215, 1960.
4. Davis, M.; Logemann, G.; Loveland, D. W. A machine program for theorem-proving. Commun. ACM, v. 5, n. 7, p. 394–397, 1962.
5. Ercsey-Ravasz, M.; Toroczkai, Z. The Chaos Within Sudoku. CoRR, abs/1208.0370, 2012.
6. Felgenhauer, B.; Jarvis, F. Enumerating possible Sudoku grids. 2005. Disponível em: `http://www.afjarvis.staff.shef.ac.uk/sudoku/`.
7. Felgenhauer, B.; Jarvis, F. Mathematics of Sudoku I. 2006. Disponível em: `http://www.afjarvis.staff.shef.ac.uk/sudoku/`.
8. Geem, Z. W. Harmony Search Algorithm for Solving Sudoku. In: . KnowledgeBased Intelligent Information and Engineering Systems: 11th International Conference, KES 2007, XVII Italian Workshop on Neural Networks, Vietri sul Mare, Italy, September 12-14, 2007. Proceedings, Part I. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 371–378.
9. Jain, S., Shakher, C. Mathematical and C Programming Approach for Sudoku Game. Journal of Game Theory, v. 3, n. 1, p. 1–6, 2014.
10. Lee, W. Programming Sudoku. New York: Apress, 2006. (Technology in action).
11. Lynce, I.; Ouaknine, J. Sudoku as a SAT Problem. In: In Proc. of the Ninth International Symposium on Artificial Intelligence and Mathematics: Springer, 2006.
12. Mantere, T.; Koljonen, J. Solving, Rating and Generating Sudoku Puzzles with GA. In: IEEE Congress on Evolutionary Computation: IEEE, 2007. p.1382–1389.
13. McGuire, G., Tugemann, B. and Civario, G. There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration. Experimental Mathematics, v. 23, n. 2, p. 190–217, 2014.
14. Moraglio, A.; Chio, C. D.; Poli, R. Geometric Particle Swarm Optimisation. In: Genetic Programming: 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 125–136.
15. O'Sullivan, B.; Horan, J. Generating and Solving Logic Puzzles Through Constraint Satisfaction. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada, 2007. p. 1974–1975.

16. Silva, F. da; Melo, A. de; Finger, M. Lógica para Computação. São Paulo: Thomson Pioneira, 2006.
17. Simonis, H. Sudoku as a Constraint Problem. In: In Proc. 4th Int. Workshop on Modelling and Reformulating Constraint Satisfaction Problems, 2005. p. 13–27.
18. Yato, T.; Seta, T. Complexity and completeness of finding another solution and its application to puzzles. IEICE Trans Fundam Electron Commun Comput Sci (Inst Electron Inf Commun Eng), E86-A, n. 5, p. 1052–1060, 2003.
19. Zhai, G., Zhang, J. Solving Sudoku Puzzles Based on Customized Information Entropy. International Journal of Hybrid Information Technology, v. 6, p. 77-92, 2013.

Evolving Negative Application Conditions

Andrei Costa, Rodrigo Machado and Leila Ribeiro

Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
Email: {acosta, rma, leila}@inf.ufrgs.br

Abstract. Graph grammars are formal, graph-based modeling languages that provide at the same time a visual interpretation for systems and a precise definition for concepts such as conflicts between transformations. Graph grammars are commonly extended with concepts that make the expression of certain patterns more convenient. One example is the use of negative application conditions in rules (NACs), which allow the modeler to express that a given rule should not be applied under a given circumstance. Another example of extension to graph grammars is the framework of second order graph grammars (SOGGs). They were proposed to represent programmed evolution of models and introduce second-order rules that modify the structure of (first-order) graph transformation rules. However, the current definition of second-order rule application does not support the transformation of (first-order) rules that are equipped with NACs. This is an important limitation since NACs are widely used in practice. In this paper, we give an introductory approach to integrate the framework of second order graph grammars considering graph grammars with negative application conditions in rules.

1 Introduction

Graph grammars [2] are a rule-based language for modeling complex systems [9]. The main idea is that of a graph, representing the system state, being modified by a set of rewriting rules through transformations. This feature provides a visual and intuitive modeling of the system, while a precise semantics allows the development of several analysis techniques on these grammars [3].

The state modifications are based on finding a match for the left-hand side of a rule in the state and replacing it by the induced rule modification. However, this process can be increased with some additional features, for example, negative application conditions (NACs). Those are a very useful expressive tool to model situations forbidden in a transformation. The transformations are only allowed when the state do not have the forbidden elements specified by the NACs of that production.

Another common task in systems is evolution. A graph grammar modeler can do it manually, however the effects of the embedded modifications can become very complex. To model a programmed evolution would be useful for the development of analysis techniques. In [7], system evolutions are modeled as

productions of higher level order. However, to be useful, this methodology must consider features that are frequently used in practice.

Nowadays there are many tools to model graph grammars, each with a different focus. Support for programmable evolution is, however, still very limited. The Verigraph [1] tool, which is used to support this work, provides a framework to work with graph transformation systems with support for second order transformations.

In this work we propose an extension of the second order framework that adds mechanisms to evolve NACs on an evolution of productions. Particularly, a NAC can be evolved by two different reasons: (1) the NAC evolution is induced by the production evolution; and (2) that NACs can be created and deleted. Both proposals should coexist in order to comprehend the entire second order transformation with first-order NACs definition.

This text is organized as follows: the section 2 shows the basis of first order graph grammars; the section 3 analyses the modifications on NACs when evolving productions; the section 4 introduces the evolution of NACs in the context of second order productions; and the section 5 concludes this paper.

2 Graph Grammars

In this section we review the basic definitions of algebraic graph transformation according to the double pushout approach [4]. The definitions that follow are standard in the area and can be found in books such as [3].

Definition 1 (graph) *A graph $G = (V, E, s, t)$ consists of a set V of nodes (vertices), a set E of edges, and two functions, $s, t : E \to V$, the source and target functions.*

Definition 2 (graph morphism) *Given two graphs, $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$, a graph morphism $f : G_1 \to G_2$ is a pair (f_V, f_E) of two total functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$.*

Definition 3 (typed graph) *A typed graph G^{TG} is a graph morphism $t : G \to TG$, where the source graph G is interpreted as an* instance graph *and the target graph TG is interpreted as a* type graph*. The morphism t itself is referred as a* typing morphism*.*

Definition 4 (typed graph morphism) *Given two typed graphs G_1^{TG}, G_2^{TG} with respective typing morphisms $type_1 : G_1 \to TG$ and $type_2 : G_2 \to TG$, a typed graph morphism is a triple $(type_1, type_2, f)$ where $f : G_1 \to G_2$ is a graph morphism, such that $type_2 \circ f = type_1$.*

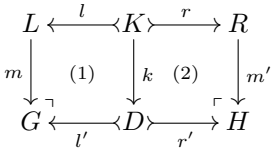
The category with typed graphs as objects and typed graph morphisms as morphisms is known as $\mathbf{Graphs}_{TG}$.

Definition 5 (typed graph production) *A typed graph production $p = L \xleftarrow{l} K \xrightarrow{r} R$ is a span of typed graph morphisms l and r, such that l and r are monomorphisms (injective). Productions are also referred as (typed) graph rules or (typed) rules.*

The L, K and R typed graphs are known as left-hand, interface and right-hand graphs, respectively. It may be the case that the same production can be applied in various ways over the same typed graph.

The definition of transformation (in the DPO approach) uses two categorical operations called *pushout* and *pushout complement*. The $\mathbf{Graphs}_{TG}$ category is closed under *pushouts*. However, *pushout complements* do not exist for all diagrams.

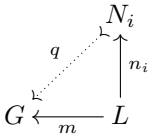
Definition 6 (match, typed graph transformation) *Given a production $p = L \xleftarrow{l} K \xrightarrow{r} R$ and a typed graph G, as in the diagram below. A match is an arbitrary typed graph morphism from L to G. A typed graph transformation $G \xRightarrow{p,m} H$ from G to H exists if the diagram below can be constructed, where (1) and (2) are* pushouts *in the category* $\mathbf{Graphs}_{TG}$.



The applicability of a production p over graph G using match m is given by the existence of morphisms k and l' such that square (1) is a *pushout*. The pair (k, l') is called a *pushout complement* of (l, m), and there are two conditions that must hold to ensure its existence. The *dangling condition* fails when the match m deletes a node and does not delete all incident edges to this node. The *identification condition* fails when the match m identifies a deleted element with any other deleted or preserved element. If both the *dangling* and *identification* conditions *succeed*, then there is a pushout complement for (l, m).

Productions can also be incremented with a set of NACs [5]. A NAC is a typed graph morphism from left or right side of a production to a typed graph with the forbid elements. A NAC is satisfied if it does not disable the transformation.

Definition 7 (NAC, production with NACs) *A negative application condition $NAC(n)$ is an arbitrary typed graph morphism $n : L \to N$. A NAC $n : L \to N$ is satisfied with respect to a match $m : L \to G$, written $G \vDash NAC(n)$, if and only if $\nexists q : N \to G$ such that q is injective and $q \circ n = m$. A production with NACs (p, NAC_p) is composed by a production p and a set of NACs (NAC_p). A match $m : L \to G$ satisfies NAC_p if and only if it satisfies all single NACs $m \vDash NAC(n_i), \forall i \in I$.*



Definition 8 (Typed graph grammar with NACs) *A typed graph grammar with NACs is a tuple (TG, G_0, P) where TG is the type graph, G_0 (initial graph) is a typed graph over TG and P is a set of productions with NACs.*

Example 1 (Pacman grammar). Figure 1 shows a graph grammar (TG, G_0, P) that models a simplified version of the pacman game, where $P = \{\text{movePacman}, \text{moveGhost}, \text{killPacman}, \text{killGhost}, \text{getBerry}, \text{dropBerry}\}$. The type graph 1(b) allows four kind of nodes: *ghost*, *pacman*, *berry* and *block* (filled circle). The edges indicate the position of the elements: *ghosts*, *pacman* and *berries* can be found in *blocks*, and *pacman* can carry *berries*. The initial graph, in 1(a), is an arbitrary typed graph specifying the initial state. The productions are shown in the remaining figures, each of them depicted by their three typed graphs (L, K and R, in this order). The morphisms $l : K \to L$ and $r : K \to R$ between the graphs are omitted but can be recovered from the position of the elements in each figure. In this simplified pacman game, *ghosts* and *pacmans* move freely over the *blocks*, according to rules 1(c) and 1(d). The *pacman* can obtain a *berry*, and occasionally drop it in anywhere, as shown in rules rules 1(g) and 1(h). When a *ghost* and a *pacman* are on the same *block*, two rules may be applied. If the *pacman* has a *berry* then it kills the *ghost*, rule 1(f). Otherwise, the *ghost* kills the *pacman*, rule 1(e). Notice that this rule requires to check if *pacman* does not have a *berry*, and therefore a NAC is necessary.

3 Evolution

Changes on graph grammars are common, specially when modeling systems that are always evolving. The capture of these changes can be very useful in terms of systems analysis. As a first approach, we will focus on production transformations. Productions can evolve in various senses: a new preserved item can be added, an item that is deleted can be preserved, a new item can be created, etc.

In order to establish a valid morphism between productions, we introduce the *rule morphism* definition [7]. The evolution is presented as a transformation from the left to the right side of the diagram.

Definition 9 (rule morphism and evolution) *A rule morphism between productions $P = (l, r)$ and $P' = (l', r')$ is a triple (f_l, f_k, f_r) of typed graph morphisms between the corresponding graphs of two rules, such that the diagram below commutes. A rule morphism is mono/epi/isomorphic if their three morphisms are also.*

$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ \downarrow{\scriptstyle f_l} & = & \downarrow{\scriptstyle f_k} & = & \downarrow{\scriptstyle f_r} \\ L' & \xleftarrow[\ l'\ ]{} & K' & \xrightarrow[\ r'\ ]{} & R' \end{array}$$

An evolution is a span of monomorphic rule morphisms.

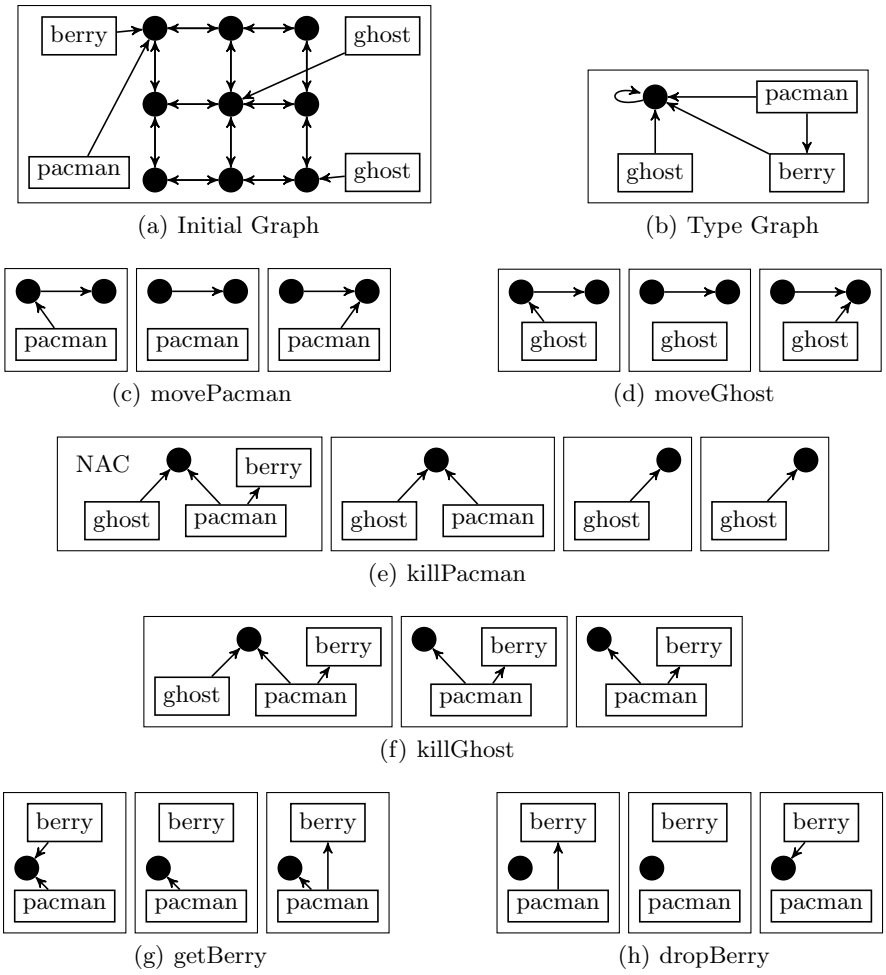
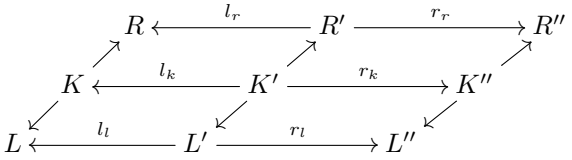


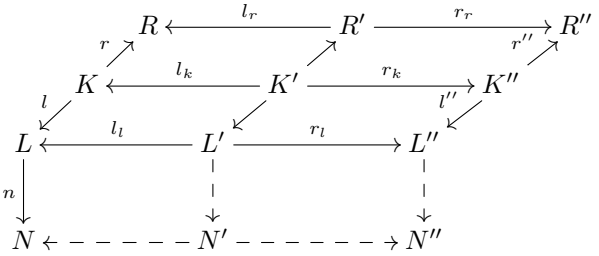
Fig. 1. Example of Graph Grammar - Pacman



In [8], an algorithm to analyse conflicts before and after evolutions was proposed. This algorithm gives a way to analyse the impact of the evolution to the system before it occurs. However, these techniques are not considering the case when the evolved production has NACs. It is very important to model this situations too, because NACs are a widely utilized model feature. We propose an

extension for evolution considering NACs on the left-hand of the left production (NACs on right-hand side can be shifted to the left-hand).

Definition 10 (evolution with NACs) *An evolution with NACs is a span of monomorphic rule morphisms, where the left-hand rule has a NAC:* $\exists n : L \to N$.



Example 2 (Evolution of the rule killPacman*).*

Suppose an evolution on the rule *killPacman* 1(e) that adds an extra pre condition to this rule, that is two *ghosts* are needed to kill a *pacman*. The NAC of the original rule forbids the applying when the *pacman* has a *berry*, in this case, the evolved NAC must forbid the same situation, but considering two *ghosts*, as well as the pre condition.

In the Figure 2, this evolution is presented. Note that it is evolving from top to bottom. The names of the graphs represent their objects in the evolution diagram, the morphisms are omitted due to lack of space but can be deduced from the position of the elements.

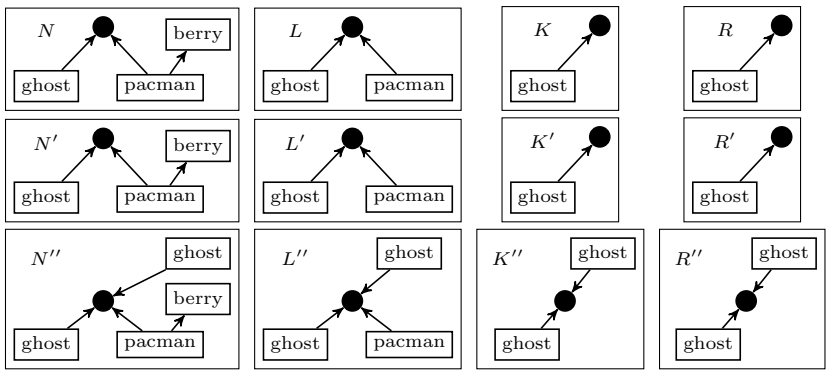


Fig. 2. Evolution with NACs Example

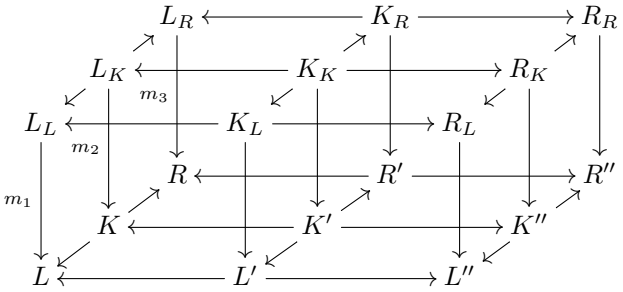
We define that the evolution of a NAC ($N \leftarrow N' \rightarrow N''$) is induced by the DPO transformation of the production ($L \leftarrow L' \rightarrow L''$) with match n. However, this transformation may not exist, in these cases we consider the resulting NAC as *true* (always satisfied). We interpret it as: the forbidden situations by N are not possible to occur with the new left-hand graph L''.

4 Second Order Productions

Second order productions, as defined in [7], model transformations of productions (without NACs). Since they are defined over DPO diagrams, the mechanics of them are almost identic to the first order productions. However, some divergent details are highlighted below.

In order to maintain the validity of the first-order productions, this transformation cannot: (i) delete some element in K, but not in L or R, in that case a additional gluing condition was proposed, called *dangling span*; and (ii) generate some non-monomorphic production, to forbid this situations a set of *minimal safety second order NACs* are automatically added to each second order production.

Definition 11 (second order transformation) *Given a span of rule morphisms* $(L_{\{L,K,R\}} \leftarrow K_{\{L,K,R\}} \rightarrow R_{\{L,K,R\}})$ *and a first order production* $p = (L \leftarrow K \rightarrow R)$*, as in the diagram below. A second order match is a monomorphic rule morphism from* $L_{\{L,K,R\}}$ *to* p*. Let* $l = (L_L \leftarrow K_L \rightarrow R_L)$*,* $k = (L_K \leftarrow K_K \rightarrow R_K)$ *and* $r = (L_R \leftarrow K_R \rightarrow R_R)$*, a second order transformation is defined in the diagram below, where* $L \overset{l,m_1}{\Longrightarrow} L''$*,* $K \overset{k,m_2}{\Longrightarrow} K''$ *and* $R \overset{r,m_3}{\Longrightarrow} R''$ *are typed graph transformations and* $(L' \leftarrow K' \rightarrow R')$ *and* $(L'' \leftarrow K'' \rightarrow R'')$ *are valid typed graph productions.*



Example 3 (Second order transformation).

Considering a second order production that has in the pre condition a rule that preserves a *ghost* in a *block*, and as pos condition it adds a new preserved *ghost* on the same *block*. Figure 3 shows a simplified view of this second-order production, where the first-order productions are collapsed.

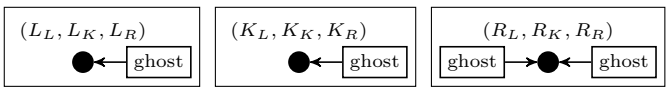


Fig. 3. Second-Order Production that adds a preserved *ghost*

This second order production has a match on the rule *killPacman* 1(e), this transformation is the same that was presented in the Example 2.

4.1 Negative Application Conditions Evolution

The down plane of the second order transformation diagram is an evolution as presented in the section 3. In that section we defined how the NACs on L would evolve, but when we work with second order productions (that induces production transformations) we are able to define transformations of the set of NACs. However, dealing with that situation can be very complex, for that, we divide it in three cases, when the second order production: creates NACs; deletes NACs; and changes NACs. All the definitions in this section point to extensions on the second order transformation diagram.

Create Considering the second order transformation diagram, we model a creation of NAC on the left-hand of the right side of the second order production (R_L). It means that a second order transformation creates NACs, as Figure 4 shows. However, we need to translate the modeled NAC, on R_L, to the left-side of the generated first order production (L''), this operation may return not an unique NAC, but a set of NACs. It occurs because the forbidden elements of a NAC can be collapsed with the elements of L'' that are not in R_L, and all this potential collapsing can not be captured by only one NAC. This operation is known in the literature as shift NAC over morphism [6].

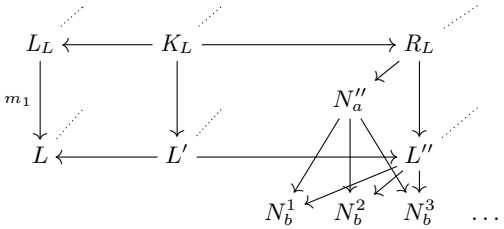


Fig. 4. Second Order Transformation Adding NACs

Example 4 (Creating NACs).

An example of that occurs on pacman grammar. Following the diagram in the Figure 4 and the graphs in the Figure 5. Considers a second order production that all objects are empty, except by a creation NAC (N''_a). Perhaps that it can be applied in various first-order productions, because empty graphs have morphism to any other graph.

Suppose that it is applied on a production that has as pre-conditions the graph L (*movePacman*). The set of created NACs N_b must forbid all situations where N''_a occurs, but it can not assume that the *pacman* in L is the same that in N''_a. In this way, 12 NACs must be created to capture all possible forbidden situations, we show only the less and the most collapsed in N^1_b and N^{11}_b, respectively.

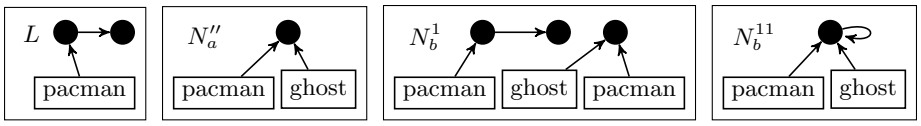


Fig. 5. Example of Second Order Transformation Adding NACs

Delete To model the deletion of NACs, a second order production must contain NACs on the left-hand of the left side (L_L). The second order transformation can be very straightforward setting all NACs matched as *true*. However, is necessary to define how is the match between second order NACs and first order NACs. Basically, it can be done by an extension of the definition of *rule morphism*. As consequence, once this NACs are on the left side of the second order production, they are included on pre-conditions of the second order production.

To define this match is not simple, suppose that the second order production has two NACs, as in Figure 6, we have two possibilities for matches from N_1+N_2 to $N'_1 + N'_2 + N'_3$: (1) to set them as arbitrary morphisms, and then we could have unexpected deletions, because it will can occur for all NACs N'_j which have the subgraph that N_i forbids; and (2) to set that these matches needs to be isomorphic out of the image n_i, thus we force that only exactly as second order production NACs (at least by the L_L subgraph) can be matched, and consequently if two NACs are matched, they are isomorphic.

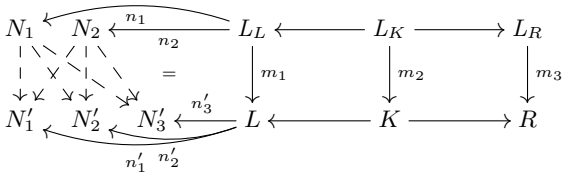


Fig. 6. Match of first order productions with NACs

Modify Differently that was proposed early in this work, by modify we mean a programmed semantic modification of the NAC, it differs for transpose a NAC without change their forbid situations, as proposed in the section 3. In this context, we could generalize creation and deletion into a unique system, that also supports transposing and modifications. However, we need to formalize a very complex system to deal with it, in which their benefits are non usual when evolving graph grammars, change the NAC semantic is not usual, we think that delete and create sequentially is the nearest from the usual modeling flow of systems with NACs.

5 Conclusions

This paper presents an ongoing work on an extension of the second order transformation. Particularly, the evolution of the first-order NACs proposed allows three characteristics: to maintain the forbidden situations; to model new forbidden cases; and to model disabling of forbidden cases.

The evolution with NACs diagram is implemented in Verigraph tool, which facilitates tests and generation of examples for our study. However, we still must prove that the translated NACs forbid the same situations (in a modified context) of the original NACs. Furthermore, we must also verify that, when there is no DPO transformation, to set as *true* the NAC is a valid semantic operation.

The extension for the second order production definition is in current work. The new definitions must match the expected behavior when modeling second order productions. For example, in current definitions, a second order production that creates a new NAC, depending of the context, can generate more than one NAC in a transformation. Another example occurs on deletion, that depending of the *rule morphism* definition, it can have a very different semantic operation.

Acknowledgments

The authors would like to acknowledge the brazilian agency CNPq for support in the form of financial aid (VeriTes project).

References

1. Andrei Costa, Jonas Bezerra, Guilherme Azzi, Leonardo Marques, Thiago Rafael Becker, Ricardo Gabriel Herdt, and Rodrigo Machado. Verigraph: a system for specification and analysis of graph grammars. *SBMF 2016*, page to appear, 2016.
2. Hartmut Ehrig. *Introduction to the algebraic theory of graph grammars (a survey)*, pages 1–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 1979.
3. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
4. Hartmut Ehrig, Michael Pfender, and Hans-Jürgen Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 167–180, Oct 1973.
5. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3, 4):287–313, 1996.
6. Leen Lambers. *Certifying rule-based models using graph transformation.* PhD thesis, Berlin Institute of Technology, 2009.
7. Rodrigo Machado. *Higher-order graph rewriting systems.* PhD thesis, Instituto de Informática - Universidade Federal do Rio Grande do Sul, 2012.
8. Rodrigo Machado, Leila Ribeiro, and Reiko Heckel. Rule-based transformation of graph rewriting rules: Towards higher-order graph grammars. *Theoretical Computer Science*, 594:1–23, 2015.
9. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

A note on bimachines

Rodrigo de Souza*

UFRPE, Recife, Brazil
rodrigo.npmsouza@ufrpe.br

Abstract. In the seventies, Schützenberger and Eilenberg introduced the notion of bimachine as a deterministic machine for the realisation of rational functions. We discuss a reworking of this classical model, which we call look-ahead bimachine. Our variant can realise any rational function with a deterministic reading of the input word, and every new state during this reading is explicitly computed by the machine with a forward scanning of the remaining of the input.

Keywords: rational function, transducer, bimachine

1 Introduction

In this communication we propose a variant of the bimachine model which, as the bimachines, is powerful enough to realise the whole family of rational functions; however, contrary to the classical model, the computations of our machine consist of explicit sequences of forward and backward moves over the input word.

Transducers, or automata with outputs, are two-tape automata which realise relations betweens words; the family of such relations, the *rational relations*, forms one of the cornerstones of the theory of automata, as Eilenberg wrote in his treatise [6]. As classical automata, a transducer reads input words from left to right, but every transition bears, besides the letter being read, an output word: in every successful computation reading the word u, the concatenation of these outputs is a word in the image of u (in the relation realised by the transducer).

Besides their mathematical richness, transducers have been applied extensively as models and algorithms for real-world problems, in different domains, such as natural language processing [11], image processing [1, 5] and bioinformatics [3]. A modern account of the subject, as well as a reference for the terminology and concepts which appear freely, with no previous formal definition in this text (due to space constraints), can be found in Jacques Sakarovitch's book [13].

A particular rôle is played by the single-valued, or *functional*, transducers, that is, the ones which realise word-to-word functions – the *rational functions*. The rational functions are the subject of this communication, and the problem addressed here is the conception of deterministic models for the realisation of such functions. The interest of this question lies in the fact that, unlike classical automata, functional transducers cannot be determinised. This comes from

* This research is supported by the project *Problemas estruturais em modelos formais de Computação*, Edital MCTI/CNPq/Universal 2014.

a deep result of Choffrut [4], which characterises the *sequential functions*, the ones which can be realised by a *sequential*[1] transducer (= the underlying input automaton is deterministic). In other words, not every rational function can be realised by a deterministic, sequential left-to-right reading of the input word. A very simple example is the function f over the two-letter alphabet $A = \{a, b\}$ which, for every $u \in A^+$, strips the trailing a's of u, that is:[2]

$$uf = \begin{cases} 1 & \text{if } u \text{ has no occurrence of } b\text{'s} \\ xb & \text{if } u \text{ is of form } xba^n \end{cases} \tag{1}$$

This functions can be realised by a small (four-states) transducer, but by Choffrut's Theorem it is not sequential.

Thus, different machineries must be developed if one wants to computes (the images of) a rational function with a deterministic reading of the input word. There are two classical approaches in the literature; one of them is the following theorem due to Elgot and Mezei [7], which combines two kinds of sequential transducers or, more precisely, the functions realised by them: *left sequential transducers* – these are the aforementioned sequential transducers, which read the input words from left to right – and the *right sequential* ones, where the deterministic reading of the input word is from right to left:

Theorem 1 (Elgot-Mezei 1965). *A function from a free monoid A^* to a free monoid B^* is rational if, and only if, it is the composition of a left sequential function with a right sequential function.*

Elgot-Mezei's Theorem says that for every rational function f, from A^* to B^*, one can define two sequential machines: the first one reads deterministically, from left to right, input words in A^* and, for every $u \in A^*$, it writes a word x in some intermediate free monoid C^*; the second machine reads x *from right to left*, also deterministically, and writes a word in B^*. The composition of both functions is f.[3] Such decomposition can be constructed effectively from a transducer $\mathcal{T}$ realising f. Indeed, the original proof is structured as follows: the left-to-right transducer outputs, after reading u, a description of the successful computations of $\mathcal{T}$ reading u – this is the intermediate word x (in this description, the letters of C represent sets of transitions of $\mathcal{T}$); next, the right-to-left reading of x writes the outputs of precisely one of the computations encoded in this word.

An older construction was proposed by Schützenberger in 1961 [16], and later reworked by Eilenberg in [6], where it is called *bimachine*:

[1] As in [10] (see Remark 5), we prefer to call such transducers *sequential*, avoiding the traditional term *subsequential*.

[2] The empty word is represented by 1; it is the unity element of the free monoid A^*. Also note that we use the prefix notation for functions: the name of the function is written after the argument.

[3] The other direction follows from a more general property: the rational relations are closed by composition.

Theorem 2 (Schützenberger-Eilenberg 1974). *A function from a free monoid A^* to a free monoid B^* is rational if, and only if, it is the behaviour of a bimachine.*

As in Elgot-Mezei's Theorem, the definition of bimachine involves a left-to-right transducer $\mathcal{L}$ and a right-to-left one $\mathcal{R}$;[4] but here, both read somewhat simultaneously the input word $u \in A^*$. Roughly, in every step of the reading, the configuration of the bimachine consists of a position j of u and a pair (p, q) of states of $\mathcal{L}$ and $\mathcal{R}$ respectively. The machine outputs a word, which depends on (p, q) and the letter a being read, goes to position $j+1$ and changes the states as follows: p goes to $p \cdot a$ – the new state of $\mathcal{L}$ after the reading of a, and q goes to $q' = x \cdot i$, where i is the initial state of $\mathcal{R}$, x is the suffix of u starting at position $j + 1$, and $x \cdot i$ is the left action representing the right-to-left reading of $\mathcal{R}$.

Of course one can describe, by induction on the length of u, the sequence of these configurations, and thus the corresponding output word. But this is a purely formal expedient, not a mechanical one: the new states q' of $\mathcal{R}$ are not explicitly computed by a sequence of deterministic moves of the bimachine.

See Section 2 for the formal definition. Details can be found in [2] or [6]. For more recent work (applications and extensions) on bimachines, see [12, 15].

Our main motivation to introduce a new form of deterministic realisation for the rational functions stems from the remark that in both classical constructions the reading of the input word u does not consist of a pure sequence of deterministic moves over u, for some extra resource, so to speak, is involved: in Elgot-Mezei's Theorem a new word, different from the input word, must be read, and in the bimachine construction, the computation of the new states of $\mathcal{R}$ does not correspond, as we said, to an explicit sequence of moves – this would require a rewind to the right edge of the word, followed by the sequence of moves of $\mathcal{R}$ from right to left until position (which must be found in some way) $j + 1$.

In our proposal, which we call *look-ahead bimachine* – LAB for short – both issues are addressed in the following way: the reading of the input word u proceeds from left to right, and it is a sequence of "look-aheads"; each look-ahead is a deterministic sequence of moves which scans some special prefix of the remaining of the input word; these moves (more precisely, their outputs) are all the machine needs to compute the output word. Thus the main result of this communication is the following statement:

Theorem 3. *A function from a free monoid A^* to a free monoid B^* is rational if, and only if, it is the behaviour of a look-ahead bimachine.*

Before going into the details, let us note that Theorem 3 can be seen as a specialisation for rational functions of the following property of *two-way transducers* established by Engelfriet and Hoogeboom [8] and, what is more important for us, of a new, more structural proof for it we presented in [17]:

Theorem 4 (Engelfriet-Hoogeboom 2001). *Functional two-way transducers can be effectively turned into equivalent sequential two-way transducers.*

[4] Both concepts are indeed related. In [2], bimachines are used to prove Elgot-Mezei's Theorem.

As we said, such a conversion does not hold for *one-way* transducers. Our proof for Theorem 4 consists of a construction we call *pathfinder transducer*: a sequential two-way transducer which simulates deterministically the moves of a functional two-way transducer $\mathcal{T}$. Starting from $\mathcal{T}$, we define two sequential two-way transducers, a *left* and a *right pathfinder* in our terminology. Left pathfinders have been implicitly defined by Hopcroft and Ullman to prove that languages realised by certain automata (two-way balloon automata) are closed under the inverse of left sequential functions [9]. In [17], left and right pathfinders are used together to find, for every input word u, the sequence of moves of a successful computation of $\mathcal{T}$ reading u (and thus the corresponding output word).

Maybe the best way to grasp the definition of the look-ahead bimachine model is to put it in parallel with the classical notion we have just explained. Like bimachines, a LAB can be seen as a combination of two automata, a left-to-right $\mathcal{L}$ and a right-to-left $\mathcal{R}$; the output function is defined for pairs of states as well. But it has an additional component: a left pathfinder, the aforementioned construct we used to prove Theorem 4. This is what allow it to "discover" the new configuration, the one it must go for the reading of position $j+1$.

More precisely, for every position j of the input word, the LAB does a "look-ahead": a deterministic scanning of the suffix starting at j. This scanning is a sequence of forward moves, and next a sequence of backward ones, which comes back to j. But this departing position cannot be stored by the machine. Now, the left pathfinder comes in handy: it simulates backwards the computations of $\mathcal{R}$ until some position where the (unique) successful one can be distinguished (such a position always exist); next, it goes back to j, by using the property that the computations "meet" precisely in this position.

We also note that in [17] left pathfinders are defined over unambiguous transducers. The latter are previously constructed (starting from the original two-way transducer) with the *lexicographic covering* construction we defined in [14] to decompose finite-valued transducers. In such a covering, a larger automaton is constructed, whose computations project on those of the departing automaton, and from it one can extract an automaton containing precisely the smallest successful computations for some lexicographical ordering put on the transitions. Here, we describe a direct construction of the pathfinder (and thus of the LAB), for an arbitrary transducer, not necessarily unambiguous – the lexicographic selection of successful computations is somewhat embodied in the construction.

In the sequel, we shall present a minimum of formal notation regarding automata and transducers, and next discuss our proof of Theorem 2. Some claims (propositions), describing simple properties of the constructions being described, will be stated without proof.

2 Automata and transducers

We shall implicitly consider that every input word is surrounded by endmarks. This allows in particular to detected its initial and the final positions.

One-way automata, or simply automata, are acceptors which read the input tape from left to right; in *one-way transducers* the reading is made in two tapes, also from left to right, and pairs of words are accepted. Both consist of a finite set Q of *states*, sets I, $T \subseteq Q$ of *initial* and *final* states, respectively, and the set E of transitions. In an automaton over the alphabet A, $E \subseteq Q \times A \times Q$, that is, transitions are labelled by letters; in a transducer over the alphabets A and B, the labels are pairs in $A \times B^*$: a letter in the first tape and a word in the second.[5] As usual, automata and transducers can be depicted as labelled directed graphs; see Figure 1 for an example.

The *behaviour* of an automaton $\mathcal{A}$ is the subset of A^* consisting of the labels of the *successful computations* of[6] $\mathcal{A}$. A computation represents the reading of a word: it is a sequence of consecutive transitions, $c : p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_\ell} p_\ell$; its label is $a_1 \dots a_\ell$, and c is *successful* if $p_0 \in I$ and $p_\ell \in T$. For transducers, the label of a computation is the pair obtained by the componentwise concatenation of the labels of the transitions, and the behaviour is a subset of $A^* \times B^*$.

From a "dynamic" point of view, the behaviour of a transducer is a relation from A^* to B^* which sends every word $u \in A^*$ to the set of words $x \in B^*$ such that (u, x) is the label of some successful computation — the *image* of u. In this setting, we say that a computation labelled by (u, x) *reads* u and *writes* x, and that u is its *input* and x its *output*. The transducer is functional is every image has at most one element.

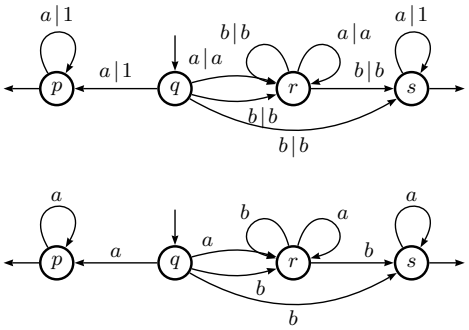


Fig. 1. A functional transducer $\mathcal{T}$ over $\{a, b\}^* \times \{a, b\}^*$ and its underlying input automaton. Initial and final states are indicated by ingoing and outgoing arrows, respectively. In $\mathcal{T}$, every transition is labelled by a pair (x, y), where x is a letter in $\{a, b\}$ and y is a word over this alphabet; such a pair is represented as $x|y$. The behaviour of $\mathcal{T}$ is the function f defined in (1).

[5] In general, transducers are labelled by pairs of words, elements of the product monoid $A^* \times B^*$; but for the *finitely-valued* transducers we are dealing with, it is not restrictive to impose that the first component is always a letter [13]. Such a transducer is called a *nondeterministic generalised sequential machine* in some references.

[6] A *rational subset* of A^*.

The notation below is the same as in [17]. Let $\mathcal{A}$ be an automaton. We denote by $\mathcal{A}_{\mathsf{det}}$ the deterministic automaton obtained from $\mathcal{A}$ by the subset construction, and by $\mathcal{A}^o_{\mathsf{det}}$ the automaton obtained by reversing $\mathcal{A}$ and applying the subset construction. For every deterministic automata, we denote by a dot the extended transition function – this is a *right action* of A^* over the states of the automaton. Thus, the state reached from the initial state I of $\mathcal{A}_{\mathsf{det}}$ with the reading of the word x is $I \cdot x$. Symmetrically, see $\mathcal{A}^o_{\mathsf{det}}$ as an automaton which reads from right to left[7] starting at its initial state J, and for every $y \in A^*$ denote by $y \cdot J$ the state of $\mathcal{A}^o_{\mathsf{det}}$ reached with the reading of the reversal of y. Now, $\cdot$ is a *left action* of A^* over the states of $\mathcal{A}^o_{\mathsf{det}}$. The following proposition is a simple property of deterministic automata which relates the subsets $I \cdot x$ and $y \cdot J$ to computations of $\mathcal{A}$:

Proposition 1. *For every word x, $I \cdot x$ is the set of ends (states of $\mathcal{A}$) of computations of $\mathcal{A}$ which start at some initial state and are labelled by x, and $y \cdot J$ is the set of states q of $\mathcal{A}$ such that there is a computation labelled by y from q to some final state.*

A *bimachine* over the alphabets A and B consists of two sets of states, P and Q, two initial states, $i \in P$ and $j \in Q$, a right and a left action of A^* over P and Q, respectively (the transition functions), and an "output function" $\gamma : Q \times A \times P \to B^*$, which can be extended to a function $Q \times A^* \times P \to B^*$ by: $(q, 1, p)\gamma = 1$, $(q, xa, p)\gamma = (q, x, a \cdot p)\gamma(q \cdot x, a, p)\gamma$, for $a \in A$, $x \in A^*$. For every input word $u \in A^*$, the image of u by (the function realised by) the bimachine is the word $(i, u, j)\gamma \in B^*$. The mechanical description of the reading of u sketched in the Introduction is equivalent to the following formal fact, which can be established by induction on the length of u: put $u = a_1 a_2 \dots a_n$, where every a_i is a letter; the output of u is the concatenation

$$(i,\, a_1,\, a_2 a_3 \dots a_n \cdot j)\,\gamma\,(i \cdot a_1,\, a_2,\, a_3 a_4 \dots a_n \cdot j)\,\gamma \\ \dots (i \cdot a_1 a_2 \dots a_{n-1},\, a_n,\, j)\,\gamma$$

Lexicographic orderings of computations are described in [14] as a method to construct unambiguous automata. First, an ordering $<_l$ is put in the set E of transitions of the automaton, and is extended on E^* and thus on the computations of $\mathcal{A}$ in the following way: $c = e_1 e_2 \dots e_l e_{l+1} \dots e_n$ and $d = e'_1 e'_2 \dots e'_l e'_{l+1} \dots e'_m$ ($e_i, e'_j \in E$ for $1 \le i \le n$ and $1 \le j \le m$) are such that $c <_l d$ iff c and d have the same label (thus $m = n$) and there exists l such that $e_i = e'_i$ for $1 \le i \le l-1$ and $e_l <_l e'_l$. Thus, E is viewed as an alphabet, and two computations as words over this alphabet; two computations are comparable iff they have the same label. In the look-ahead bimachine construction, for every input word exactly one successful computation of the departing transducer is "simulated": the smallest one by the ordering $<_l$.

[7] Although $\mathcal{A}^o_{\mathsf{det}}$ is a legitimate one-way automaton, which recognises the reverse of the words accepted by $\mathcal{A}$.

3 Look-ahead bimachines

Now we shall proof both directions of Theorem 3.

From transducers to look-ahead bimachines

Let $\mathcal{T}$ be a functional transducer. As we said, our LAB for $\mathcal{T}$, that we denote by $\mathcal{L}$, is based on a left pathfinder transducer, in the terminology of [17]. But in our construction, $\mathcal{T}$ is an arbitrary transducer, not necessarily unambiguous, as required in [17]. Actually, the preliminary construction of an unambiguous transducer made in [17] is embedded in the definition of $\mathcal{L}$. In other words, our construction selects, for every input letter being read, the smallest transition of $\mathcal{T}$ according to a lexicographic ordering $<_l$ put on the set of computations.

Let $\mathcal{A}$ be the underlying input automaton of $\mathcal{T}$. The LAB $\mathcal{L}$ has two sets of states: the *output states*, and the *scanning states*.

The former consists of pairs (R, S) of states of $\mathcal{A}_{\mathsf{det}}$ and $\mathcal{A}^{o}_{\mathsf{det}}$, respectively; only in transitions starting at these pairs the transducer writes an output. As in classical bimachines, when scanning the position j of the input word $u = a_1 \dots a_\ell$, the configuration of $\mathcal{L}$ is a triple

$$(R_j, a_j, S_j)$$

and the main invariant of the computations of $\mathcal{L}$ is that it represents particular sets of states of $\mathcal{T}$:

$$R_j = I \cdot (a_1 \dots a_{j-1}), \quad S_j = (a_{j+1} \dots a_\ell) \cdot J$$

(I and J are the initial states of $\mathcal{A}_{\mathsf{det}}$ and $\mathcal{A}^{o}_{\mathsf{det}}$, respectively).

The reading of the letter a_j consists in writing an output word and constructing the new configuration $(R_{j+1}, a_{j+1}, S_{j+1})$. For the first component, $\mathcal{L}$ simply follows a transition of $\mathcal{A}_{\mathsf{det}}$:

$$R_{j+1} = R_j \cdot a_j$$

The trick part is the construction of S_{j+1}, which we explain at the end of this section. Let us at first describe how outputs are written. It depends of a state

$$q_j \in R_j \cap a_j \cdot S_j$$

which is also stored by $\mathcal{L}$. Its meaning is expressed in the following proposition:

Proposition 2. *The states in $R_j \cap a_j \cdot S_j$ are exactly the ends of the prefixes labelled by $a_1 \dots a_{j-1}$ of the successful computations of $\mathcal{T}$ labelled by $a_1 \dots a_\ell$.*

Thus, q_j is the end of one of these prefixes, and the construction of q_{j+1} from q_j must assure that the resulting computation is the smallest one according to the lexicographic ordering $<_l$. In other words, $\mathcal{L}$ must chose j-th the transition of $\mathcal{T}$ of the computation being simulated. This choice is based on the set

$$X = R_{j+1} \cap S_j$$

which by Proposition 2 is the set of ends of prefixes labelled by $a_1 \dots a_j$ of the successful computations of $\mathcal{T}$. Now, the transition to be taken is the smallest one according to $<_l$ in the set

$$\{q_j \xrightarrow{a_j|x} q : q \in X\}$$

The end of this transition is the new state q_{j+1} and its output is written by $\mathcal{L}$.

Now we explain the construction of the new set S_{j+1}. If the input word u is in the domain of $\mathcal{T}$, then in $\mathcal{A}^o_{\mathsf{det}}$ there are transitions of form

$$X_1 \xrightarrow{a_{j+1}} S_j, \dots, X_t \xrightarrow{a_{j+1}} S_j$$

and exactly one of the X_i's is S_{j+1}. It may happen that there is exactly one $X_i \xrightarrow{a_{j+1}} S_j$, and then $S_{j+1} = X_i$. Otherwise, $\mathcal{L}$ enters the scanning set of states to "discover" the aimed X_i – these states correspond to the left pathfinder construction, which we recall below.

The bimachine $\mathcal{L}$ scans a prefix of $a_{j+1} \dots a_\ell$, in two parts: at first, it performs a sequence of forward moves, and next, a sequence of backward moves, which allows to come back to position j. Let us at first explain the forward moves made by $\mathcal{L}$. They are based on the automaton $\mathcal{C}$ obtained by *reversing $\mathcal{A}^o_{\mathsf{det}}$ again and applying the subset construction.* For every X_i, $1 \le i \le t$, denote by $\mathcal{C}(X_i)$ the part of $\mathcal{C}$ accessible from X_i; consider the product of automata

$$\mathcal{C}(X_1) \times \dots \times \mathcal{C}(X_t)$$

which is also a deterministic automaton. Starting from state $X_1 \times \dots X_t$ and position $j+1$, the pathfinder simulates the deterministic $\mathcal{C}(X_1) \times \dots \times \mathcal{C}(X_t)$, that is: it scans successively the positions $j+2, j+3, \dots$ of the input word u and visits the states

$$(X_1 \cdot a_{j+2}) \times \dots \times (X_t \cdot a_{j+2}), \quad (X_1 \cdot a_{j+2}a_{j+3}) \times \dots \times (X_t \cdot a_{j+2}a_{j+3}), \dots \qquad (2)$$

Here we can state the main property which allows to discover S_{j+1}:

Proposition 3. *Either at some position $k > j+1$ exactly one of the components of $(X_1 \cdot a_{j+2} \dots a_k) \times \dots \times (X_t \cdot a_{j+2} \dots a_k)$ is not empty, or the forward scanning reaches the end of u ($k = \ell$), and J (the initial state of $\mathcal{A}^o_{\mathsf{det}}$) is contained in exactly one of these components.*

This fact is a consequence of the unambiguity of $\mathcal{A}^o_{\mathsf{det}}$. In other words, at some point, the pathfinder reaches a special set of $\mathcal{C}$ – the nonempty component, or a component containing J – which we call the target set. The target set contains precisely the states of $\mathcal{A}^o_{\mathsf{det}}$ which reach S_{j+1} after the right-to-left reading from position k to $j+1$ in $\mathcal{A}^o_{\mathsf{det}}$:

Proposition 4. *Let X be the target set reached by the forward moves starting at position $j+1$, and $Y \in X$. Then, $S_{j+1} = (a_{j+2} \dots a_k) \cdot Y$.*

After finding the target set, the pathfinder goes to the sequence of backward moves. They are based on the product $\mathcal{A}^{o}_{\mathsf{det}} \times \mathcal{A}^{o}_{\mathsf{det}}$, and every pair (Y, Z) in the deterministic backward reading is such that: Y comes from an arbitrary fixed state belonging to the target set; Z comes from an arbitrary fixed state belonging to some other set (among the ones reached by the forward moves). Is follows from the fact that $\mathcal{A}^{o}_{\mathsf{det}}$ is deterministic that:

Proposition 5. *For every position greater than $j+1$, $Y \neq Z$, and $Y = Z$ in position $j+1$.*

This allows to find the position $j+1$ again, and S_{j+1}, by Proposition 4.

This closes the description of the scanning phase, but not of $\mathcal{L}$: first of all, the initial configuration (R_1, a_1, S_1) must be computed. This is easy: the set R_1 is I, and to find $S_1 = u \cdot J$, $\mathcal{L}$ goes to the right edge of u, and comes back, applying the transitions of $\mathcal{A}^{o}_{\mathsf{det}}$.

From look-ahead bimachines to transducers

Here we just sketch the proof that, given a LAB $\mathcal{L}$, one can construct an equivalent functional transducer $\mathcal{T}$. In general, the obtained transducer is not sequential: this is due to the nondeterministic "guessings" embodied in its definition.

Every state of $\mathcal{T}$ is a pair (q, P) formed by an output state q of $\mathcal{L}$ and P a function from the output states to sets of pairs of scanning states. In every computation

$$c : (q_1, P_1) \xrightarrow{a_1|x_1} (q_2, P_2) \xrightarrow{a_2|x_2} \dots \xrightarrow{a_\ell|x_\ell} (q_{\ell+1}, P_{\ell+1})$$

of $\mathcal{T}$, every (p_j, P_j) projects, in the first component, in the output state of $\mathcal{L}$ reached with the reading of $a_1 a_2 \dots a_{j-1}$. The function P_j aims to capture the forward-backward scanning of segments $a_r a_{r+1} \dots a_s$, with $r \leq j \leq s$: for every output state p, the image of p, by P_j, is the set of pairs of states reading a_j in all the possible scannings starting at p. Every pair with equal components represents the "closing" of some scanning, and is not stored.

In order to be successful, c must represent a sequence of well-formed forward-backward scannings. This means that the image of every output state by $P_{\ell+1}$ is the empty set ("all scannings have been closed"); such functions mark the final states of $\mathcal{T}$.

Acknowledgements The author thanks Wilson Rosa for his endless disposition to push people to do research in theoretical Computer Science, and the anonymous referees for their valuable remarks, which helped him to improve the manuscript.

References

1. Albert, J., Kari, J.: Digital image compression. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata, pp. 453–479. Springer

Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-01492-5_11
2. Berstel, J.: Transductions and context-free languages. Teubner Verlag (1979)
3. Bradley, R.K., Holmes, I.: Transducers: an emerging probabilistic framework for modeling indels on trees. Bioinformatics (Oxford, England) 23(23), 3258–62 (dec 2007), http://bioinformatics.oxfordjournals.org/cgi/content/long/23/23/3258
4. Choffrut, C.: Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. Theoretical Computer Science 5(3), 325–337 (Dec 1977), http://dx.doi.org/10.1016/0304-3975(77)90049-4
5. Culik, K., Kari, J.: Digital images and formal languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages: Volume 3 Beyond Words, pp. 599–616. Springer Berlin Heidelberg, Berlin, Heidelberg (1997), http://dx.doi.org/10.1007/978-3-642-59126-6_10
6. Eilenberg, S.: Automata, Languages, and Machines. Academic Press, Inc., Orlando, FL, USA (1974)
7. Elgot, C.C., Mezei, J.E.: On Relations Defined by Generalized Finite Automata. IBM Journal of Research and Development 9(1), 47–68 (1965)
8. Engelfriet, J., Hoogeboom, H.J.: MSO definable string transductions and two-way finite-state transducers. ACM Trans. Comput. Logic 2(2), 216–254 (2001), http://dx.doi.org/10.1145/371316.371512
9. Hopcroft, J.E., Ullman, J.D.: An approach to a unified theory of automata. In: 8th Annual Symposium on Switching and Automata Theory (SWAT 1967). pp. 140–147. IEEE Computer Society (1967)
10. Lombardy, S., Sakarovitch, J.: Sequential? Theoretical Computer Science 356(1-2), 224–244 (May 2006), http://dx.doi.org/10.1016/j.tcs.2006.01.028
11. Mohri, M., Pereira, F., Riley, M.: Speech recognition with weighted finite-state transducers. In: Benesty, J., Sondhi, M.M., Huang, Y.A. (eds.) Springer Handbook of Speech Processing, pp. 559–584. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-49127-9_28
12. Rhodes, J., Silva, P.V.: Turing machines and bimachines. Theoretical Computer Science 400(1-3), 182–224 (2008)
13. Sakarovitch, J.: Elements of Automata Theory. Cambdrige University Press (2009)
14. Sakarovitch, J., de Souza, R.: Lexicographic decomposition of k-valued transducers. Theory Comput. Syst. 47(3), 758–785 (2010), http://dx.doi.org/10.1007/s00224-009-9206-6
15. Santean, N., Yu, S.: Nondeterministic bimachines and rational relations with finite codomain. Fundam. Inf. 73(1,2), 237–264 (Apr 2006), http://dl.acm.org/citation.cfm?id=2369408.2369429
16. Schützenberger, M.P.: A remark on finite transducers. Information and Control 4(2-3), 185–196 (1961), http://dx.doi.org/10.1016/S0019-9958(61)80006-5
17. de Souza, R.: Uniformisation of two-way transducers. In: Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7810, pp. 547–558. Springer (2013), http://dx.doi.org/10.1007/978-3-642-37064-9_48

A Rewriting Logic Semantics for the Generalized Substitution Language

Christiano Braga[1], David Deharbe[2,3], Anamaria Moreira[4] and Narciso Martí-Oliet[5]

[1] Universidade Federal Fluminense,
[2] ClearSy System Engineering
[3] Universidade Federal do Rio Grande do Norte,
[4] Universidade Federal do Rio de Janeiro,
[5] Universidad Complutense de Madrid

Abstract. The B Method is a leading technique to specify refinements and reason about them. Essentially, refinements are captured as invariants that must hold on the transition system induced by a B machine specification. In this paper we give the first steps to the addition of temporal logic-based reachability analysis of the state transition systems induced by a B specification to the B method. We formalize the semantics of the Generalized Substitution Language, a normal form that all B specifications can be translated to. Rewriting Logic is our semantic framework of choice as not only it provides a natural framework to specify operational semantics but also the Rewriting Logic system Maude allows for many automatic verification procedures such as term search, narrowing search and model checking Linear Temporal Logic formulae.

1 Introduction

The B method [1] is one of the most popular refinement techniques to develop and reason about component-based software. Essentially, it transforms substitutions and invariants that specify a B machine, described in the Abstract Machine Notation, into propositions that must be valid. A refinement is an invariant that preserves the specification of the abstract machine. For instance, it must not be the case that the invariant of a refinement breaks the initialization of the abstract machine. The Generalized Substitution Language (GSL) is a core language in the B method where all B machine specifications can be transformed to.

Rewriting Logic [9] has been shown as a quite natural semantic framework [8] for many specification languages. The Maude system [5], an implementation of Rewriting Logic, provides many automated verification procedures such as search, narrowing, Linear Temporal Logic model checking and, more recently, Temporal Logic of Rewriting model checking [2]. As a first step towards endowing the B method with the analysis techniques available in the Maude system, we propose a Rewriting Logic semantics for GSL. This semantics captures quite naturally GSL's *operational meaning*, that is, induces a finite-state transition system, and is at the same time *symbolically executable* in Maude. Moreover, the translation from the Abstract Machine Notation to GSL can be quite directly represented by an equational theory. Therefore, a term, denoting a B

machine, can be analyzed, using the aforementioned techniques, by first being reduced to its GSL form.

The contribution of this paper is three-fold: (i) an operational semantics for GSL; (ii) a rewriting logic semantics for GSL where transitions from the operational one are identified with rewrites and (iii) a prototype executable environment for the Abstract Machine Notation in Maude.

Related work. GSL is presented in [1] with a weakest precondition (WP) semantics. Most of the work on B and GSL relies on this semantics. An interesting example is the work of Dunne in [7], which includes additional information concerning the variables in scope at a substitution to solve some delicate issues that restrict what can be stated in B due to limitations of the pure WP semantics. On the other hand, some related work proposing the embedding of B into other formalisms with strong tool support, such as Isabelle/HOL, can be found in the literature [3,6]. As in the current research, the purpose of those works is combining strengths of both worlds, to achieve further proof or animation goals. On a more theoretical line, but also with similar goals, the work in [12] proposes a prospective value semantics to define the effect of GSL substitutions on values and expressions. The meaning of a computation, specified in GSL, is given in terms of the values of an expression, if the computation is carried out. In the current paper we contribute to this line of work by discussing GSL operational semantics, its rewriting logic semantics, using Maude as the specification language for Rewriting Logic theories, and a prototype execution environment in the Maude system. We also discuss search on the narrowing relation induced by the rewrite theory that represents our GSL operational semantics.

Plan of the paper. Section 2 introduces GSL through its Structural Operational Semantics (SOS). Section 3 informally recalls the Maude language and exemplifies how narrowing works in the Maude system. Section 4 discusses the Rewriting Logic semantics framework for GSL. Section 5 concludes this paper with our final remarks and points to future work.

2 GSL structural operational semantics

The Generalized Substitution Language (GSL) is a core language in the B method where all B machine specifications can be normalized to. Its grammar is specified by the following BNF description,

$$
\begin{array}{rl}
\mathit{Subst} ::= & \texttt{skip} \quad | \\
& \mathit{Var} := \mathit{Exp} \quad | \\
& \mathit{Pred} \mid \mathit{Subst} \quad | \\
& \mathit{Subst} \; [] \; \mathit{Subst} \quad | \\
& \mathit{Pred} \Longrightarrow \mathit{Subst} \; | \\
& @ \; \mathit{Var} \; . \; \mathit{Subst}
\end{array}
$$

where *Subst* denotes the syntactical class for GSL substitutions, *Var* denotes variable identifiers, *Exp* denotes integer arithmetic expressions and *Pred* denotes Boolean expressions. The informal meaning of GSL is as follows. The `skip` construction is a GSL

program which halts normally. Simple assignment is a binary operation `x := v` that assigns the value in the second operand to the variable denoted by the first operand in the memory. The precondition substitution `P | S` behaves as substitution `S` in the current memory if predicate `P` holds and aborts otherwise. Substitution `S1 [] S2` is the non-deterministic choice between `S1` or `S2`. Guarded substitution `P ==> S` behaves as `S` if `P` is true or normally aborts otherwise. Finally, the unbounded choice `@x.S` substitution chooses some value, say n, and replaces symbol `x` by n in `S` avoiding name clashes that may arise. (Note that, although this is *logically sound*, it creates problems for its *execution* with rewriting as we have an unbounded variable on the right-hand side of the rule. We discuss this executability issue further in Section 3.)

$$Sto = (Var \mapsto_{fin} \mathbb{N})$$

$$\longrightarrow \subseteq (Subst \times Sto) \times ((Subst \times Sto) \cup \{\texttt{abort}\})$$

$$\frac{\texttt{E}, sto \longrightarrow^* n}{\texttt{v} := \texttt{E}, sto \longrightarrow \texttt{skip}, update(sto, \texttt{v}, n)} \text{ (simple)}$$

$$\frac{\texttt{P}, sto \longrightarrow^* \texttt{true}}{\texttt{P | S}, sto \longrightarrow \texttt{S}, sto} \text{ (pre 1)} \qquad \frac{\texttt{P}, sto \longrightarrow^* \texttt{false}}{\texttt{P | S}, sto \longrightarrow \texttt{abort}} \text{ (pre 2)}$$

$$\text{(bchoice 1) } \texttt{S}_1 \,[]\, \texttt{S}_2, sto \longrightarrow \texttt{S}_1, sto \qquad \text{(bchoice 2) } \texttt{S}_1 \,[]\, \texttt{S}_2, sto \longrightarrow \texttt{S}_2, sto$$

$$\frac{\texttt{P}, sto \longrightarrow^* \texttt{true}}{\texttt{P ==> S}, sto \longrightarrow \texttt{S}, sto} \text{ (guard 1)} \qquad \frac{\texttt{P}, sto \longrightarrow^* \texttt{false}}{\texttt{P ==> S}, sto \longrightarrow \texttt{skip}, sto} \text{ (guard 2)}$$

$$\text{(uchoice) } @\texttt{v}.\texttt{S}, sto \longrightarrow \texttt{S}[n/\texttt{v}], sto \quad \text{if } \texttt{v} \notin var(sto), \text{ for some } n.$$

Fig. 1. Structural operational semantics rules for GSL. Let $\texttt{v} \in Var$, $\texttt{E} \in Exp$, $sto, sto' \in Sto$, $update : Sto \times Var \times \mathbb{N} \to Sto$, $var : Sto \to 2^{Var}$, $n \in \mathbb{N}$, $\texttt{P} \in Pred$, and $\texttt{S}, \texttt{S}' \in Subst$. The expression $S[n/\texttt{v}]$ denotes the replacement of all free occurrences of variable `v` by n in substitution S. The symbol $\longrightarrow^*$ denotes the reflexive-transitive closure of relation $\longrightarrow$.

3 The Maude language

In order to model a system in Rewriting Logic, that is, to specify such a system in Maude, its static part (state structure) and its dynamics (state transitions) are distinguished. The static part is specified by means of an equational theory (many-sorted, order-sorted or membership equational logic), while the dynamics are specified by means of rules. Computation in a transition system is then precisely captured by the term rewriting relation using those rules, where terms represent states of the given system.

The distinction between the static part and the dynamic part is reflected in Maude by means of functional and system modules. Functional modules in Maude correspond to equational theories (Σ, E) which are assumed to be Church-Rosser (confluent and sort

decreasing) and terminating. Equations are used to define functions over static data as well as properties of states. Usually the equation set E is the union of a set A of structural axioms (such as associativity, commutativity, or identity), also known as equational attributes, for which matching algorithms exist in Maude, and a set E' of equations that are Church-Rosser and terminating modulo A.

System modules in Maude correspond to rewrite theories $(\Sigma, A \cup E', R)$ where rewriting with R is performed modulo the equations $A \cup E'$. Moreover, the rules R must be coherent with respect to the equations E' modulo A. Coherence means that the interleaving of rewriting with rules and rewriting with equations will not loose rewrite computations, that is, failing to perform a rewrite that would otherwise have been possible before an equational deduction step was taken. By assuming coherence, Maude always reduces to canonical form using E before applying any rule in R.

Narrowing in Maude Narrowing is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces pattern matching by unification in order to (non-deterministically) reduce these terms. Narrowing was originally introduced as a mechanism for solving equational unification problems. It was later generalized to solve the more general problem of symbolic reachability.

Example 1. Consider, for example, the following specification, borrowed from [4], for a vending machine to buy apples (a) or cakes (c) with dollars ($) and/or quarters (q). A module named `NARROWING-VENDING-MACHINE` is introduced. It declares sorts (that can be understood as sets) `Coin`, `Item`, `Marking` and `State` where the sort `Coin` is included in sort `Money`. Elements of sort `Money` are constructed by operators `empty` and by juxtaposition of coins (that is, elements of sort `Coin`). The states of the vending machine are denoted by juxtaposed money and items with operator `<_>` around it. Now, the machine behaves non-deterministically by either selling a (c)ake or an (a)pple and it the latter case also returning a coin. This is specified by rules `buy-c` and `buy-a`, respectively. The fact that four (q)uarters can be identified with a dollar ($) is specified by equation `change`.

```
mod NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ > => < M c > .
  rl [buy-a] : < M $ > => < M a q > .
  eq [change] : q q q q = $ [variant] .
endm
```

One can use the narrowing search command to answer the question: *Is there any combination of one or more coins that returns exactly an apple and a cake?* This can be done by searching for states that are reachable from one specified as a term `< M:Money >` and match the desired pattern at the end.

```
1 Maude> (search [1] in NARROWING-VENDING-MACHINE : < M:Money > ~>* < a c > .)
2   Solution 1
3   M:Money --> $ q q q
4   No more solutions.
```

Narrowing executability requirements Narrowing reachability in Maude needs the following additional requirements together with the usual executability properties for Maude modules. Let $\texttt{mod}(\Sigma, G \cup E \cup Ax, R)\texttt{endm}$ be an order-sorted system module where R is a set of rewrite rules specified with the `rl` keyword or `crl` when the rewrite rule is conditional, Ax is a set of commonly occurring axioms, E is a set of finite variant equations specified with the `eq` keyword and the attribute `variant` such that $E \cup Ax$ has a finite and complete $E \cup Ax$-unification algorithm (using the notion of variant generation [4, Section 12.9], where the equations to be used in the generation process are annotated with the `variant` attribute in the Maude specification), and G are the remaining equations specified with the `eq` or `ceq` keywords. Furthermore, the transition rules R must satisfy the following conditions:

- Conditional rules specified with the `crl` keyword are not taken into account, i.e., there may be conditional rewrite rules in the system module but they will not be used for narrowing.
- A rule's lefthand side pattern cannot be a single variable.
- The rules are $E \cup Ax$-coherent and topmost (so that rewriting is always done at the top of the term).

Then narrowing is a complete deductive method for solving existential reachability questions of the form $\exists \vec{x} t(\vec{x}) \rightarrow^* t'(\vec{x})$ in the sense that the formula holds for R if and only if there is a narrowing sequence $t \rightsquigarrow^*_{\mathcal{R}, E \cup Ax} u$ such that u and t have an $E \cup Ax$ -unifier.

4 GSL semantics in Rewriting Logic

In this section we propose a conditional rewriting logic semantics for GSL (see Fig. 2). We interpret the transition relation as a rewrite relation. In essence, for each transition rule t in the SOS semantics there exists an homonymous conditional rewrite rule in rewriting logic semantics, such that

$$\frac{P}{\gamma \longrightarrow \gamma'}\ (t) \qquad \mathbf{crl}[t] : \gamma \Rightarrow \gamma' \ \mathbf{if}\ P$$

where γ, γ' are configurations of the SOS specification, and P represents the premisses of t. The deterministic fragment of GSL (simple assignment, pre-condition and guarded substitution) is specified as equations whilst rules specify the non-deterministic one (bounded and unbounded choice). Let $\mathcal{S}$ be an equational theory representing sets Sto, $\mathcal{X}$ be an equational theory representing the evaluations of GSL expressions in the context of a store, and $\mathcal{P}$ an equational theory representing the evaluation of GSL predicates in the context of a store.

Expressions
eq $[expr1] : (\mathtt{v} + \mathtt{E}, (\mathtt{v} \mapsto n)sto) = (n + \mathtt{E}, (\mathtt{v} \mapsto n)sto)$
eq $[expr2] : (n, sto) = n$

Simple assignment
ceq $[simple] : (\mathtt{v} := \mathtt{E}, sto) = (\mathtt{skip}, sto')$ **if** $(\mathtt{E}, sto) = n \wedge sto' := update(sto, \mathtt{v}, n)$

Pre-condition
ceq $[pre1] : (\mathtt{P} \mid \mathtt{S}, sto) = (\mathtt{S}, sto)$ **if** $(\mathtt{P}, sto) = \mathtt{true}$
ceq $[pre2] : (\mathtt{P} \mid \mathtt{S}, sto) = \mathtt{abort}$ **if** $(\mathtt{P}, sto) = \mathtt{false}$

Bounded choice
rl $[bchoice] : (\mathtt{S}_1 \; [] \; \mathtt{S}_2, sto) \Rightarrow (\mathtt{S}_1, sto)$

Guarded substitution
ceq $[guard1] : (\mathtt{P} ==> \mathtt{S}, sto) = (\mathtt{S}, sto)$ **if** $(\mathtt{P}, sto) = \mathtt{true}$
ceq $[guard2] : (\mathtt{P} ==> \mathtt{S}, sto) = (\mathtt{skip}, sto)$ **if** $(\mathtt{P}, sto) = \mathtt{false}$

Unbounded choice
crl $[uchoice] : (@\mathtt{v}.\mathtt{S}, sto) \Rightarrow (\mathtt{S}[n/\mathtt{v}], sto)$ **if** $\mathtt{v} \notin var(sto)$

Fig. 2. Conditional rewriting logic semantics for GSL.

The conditional rewriting logic semantics for GSL in Fig. 2 could be directly executed in Maude if not for rule *uchoice*. Variable n is free in its right-hand side. Narrowing can be used to execute such a rule by searching for a value for n that replaces v in a (GSL) substitution S.

To cope with the requirements for narrowing executability in Maude described in Section 3, we now define a rewriting logic semantics for GSL (see Fig. 3), based on Plotkin's SCD machines [10], which is also topmost, all rules are unconditional, the pattern on the left-hand side of each rule is not a single variable and the rules are Ax-coherent, with respect to associativity and commutativity, since there are no critical pairs as equations and rules give semantics to different GSL constructions.

Proposition 1 states that conditional and unconditional rewriting logic semantics for GSL are equivalent modulo unfolding of rewrites in the condition of rules, denoted ρ.

Proposition 1. *Let $\mathcal{C}$ be the conditional rewriting logic semantics for GSL and $\mathcal{U}$ the unconditional one.*

$$\mathcal{C} \vdash t \to t' \iff \mathcal{U} \vdash t \to_{/\rho} t'$$

The intuition is that matched or ground conditions of applied equations in $\mathcal{C}$ become the first projection of a triple of sort *Conf*.[6] Their canonical form is then used to appro-

[6] Since expressions or predicates do not need context to be evaluated, a triple is sufficient to evaluate assignments, pre-conditions and guarded substitutions. A stack, as in [10], would be necessary only if expressions or predicates allowed substitutions as sub-terms that would require further contexts to be evaluated.

Expressions
eq $[expr]$: $(\mathtt{v}_1 + \mathtt{e}_1, \mathtt{v}_2 := \mathtt{e}_2, (\mathtt{v}_1 \mapsto \mathtt{n})sto) = (\mathtt{n}{+}\mathtt{e}_1, \mathtt{v}_2 := \mathtt{e}_2, (\mathtt{v}_1 \mapsto \mathtt{n})sto)$

Simple assignment
eq $[simple1]$: $(\mathtt{v} := \mathtt{E}, sto) = (\mathtt{E}, \mathtt{v} := \mathtt{E}, sto)$
eq $[simple2]$: $(\mathtt{n}_1, \mathtt{v} := \mathtt{E}, \mathtt{v} \mapsto \mathtt{n}_2\ sto) = (\mathtt{skip}, \mathtt{v} \mapsto \mathtt{n}_1\ sto)$

Pre-condition
eq $[pre1]$: $(\mathtt{P} \mid \mathtt{S}, sto) = (\mathtt{P}, \mathtt{P} \mid \mathtt{S}, sto)$
eq $[pre2]$: $(\mathtt{v}_1 = \mathtt{e}, (\mathtt{P} \mid \mathtt{S}), (\mathtt{v}_1 \mapsto \mathtt{n})sto) = (\mathtt{n} = \mathtt{e}, \mathtt{P} \mid \mathtt{S}, (\mathtt{v}_1 \mapsto \mathtt{n})sto)$
eq $[pre3]$: $(\mathtt{true}, \mathtt{P} \mid \mathtt{S}, sto) = (\mathtt{S}, sto)$
eq $[pre4]$: $(\mathtt{false}, \mathtt{P} \mid \mathtt{S}, sto) = \mathtt{abort}$

Bounded choice
rl $[bchoice]$: $(\mathtt{S}_1\ [\!]\ \mathtt{S}_2, sto) \Rightarrow (\mathtt{S}_1, sto)$

Guarded substitution
eq $[guard1]$: $(\mathtt{P} ==\!> \mathtt{S}, sto) = (\mathtt{P}, \mathtt{P} ==\!> \mathtt{S}, sto)$
eq $[guard2]$: $(\mathtt{v}_1 = \mathtt{e}, (\mathtt{P} ==\!> \mathtt{S}), (\mathtt{v}_1 \mapsto \mathtt{n})sto) = (\mathtt{n} = \mathtt{e}, \mathtt{P} ==\!> \mathtt{S}, (\mathtt{v}_1 \mapsto \mathtt{n})sto)$
eq $[guard3]$: $(\mathtt{true}, \mathtt{P} ==\!> \mathtt{S}, sto) = (\mathtt{S}, sto)$
eq $[guard4]$: $(\mathtt{false}, \mathtt{P} ==\!> \mathtt{S}, sto) = (\mathtt{skip}, sto)$

Unbounded choice
rl $[uchoice]$: $@\mathtt{p}.\mathtt{S}, sto \Rightarrow \mathtt{S}[n/\mathtt{p}], sto,$ for some n, where p is a place holder.

Fig. 3. Unconditional rewriting logic semantics for GSL.

priately replace the left-hand side by the right-hand side of a given equation, as specified by equations *simple2*, *pre2* and *guard2*.

Proof (Sketch). Rewrites in the conditions of $\mathcal{C}$ are not deductions in $\mathcal{C}$ (the so called "scratchpad rewrites"). In $\mathcal{U}$ they are "unfolded" and become deductions. Equivalence modulo unfolding of rewrites in the conditions works as follows for expressions. A similar reasoning can be applied to predicates.
($\Longrightarrow$) By structural induction. For *Conf* terms with simple assignment of the form $\mathtt{v} := \mathtt{n}, (v \mapsto k)sto$, where $n, k \in GNat$, the canonical form $skip, (\mathtt{v} \mapsto n)sto$ is reached in one step in $\mathcal{C}$ by the application of equation *simple*. The same canonical form is reached in $\mathcal{U}$ in two steps by application of equations *simple1* and *simple2*. For general simple assignments of the form $(\mathtt{v} := E, sto)$,

$$\mathcal{C} \vdash (\mathtt{v} := E, sto) \Rightarrow^{2w+1} (\mathtt{skip}, (v \mapsto n)sto),$$

where w is the number of arithmetic operations in E, and $2w+1$ denotes the maximum number of rewrites for this case, when all operands are variables. The canonical form for the general case in the unconditional rewriting logic semantics for GSL is reached in $2w+2$ steps by the application of equation *simple1*, two times w rewrites with *simple3*

and one rewrite with *simple2*.

$$\begin{array}{rl} \mathcal{U} \vdash (\mathtt{v} := E, (\mathtt{v} \mapsto k)sto) & \rightarrow \\ (E, \mathtt{v} := E, (\mathtt{v} \mapsto k)sto) & \rightarrow^{2w} \\ (n, \mathtt{v} := E, (\mathtt{v} \mapsto k)sto) & \rightarrow (skip, (v \mapsto n)sto) \end{array}$$

Terms of sort *Conf* with pre-conditions or guarded substitution follow a similar reasoning. The rule for bounded choice in $\mathcal{C}$ is not conditional so it remains the same in $\mathcal{U}$. Rule *uchoice* in $\mathcal{C}$ is conditioned to the absence of the variable v in the first parameter of $@v.S$ within the store S in its third parameter. This predicate is checked in the right-hand side of rule *uchoice* in $\mathcal{U}$ using operator *if_then_else_fi*.
($\Longleftarrow$) Rewrites on terms of sort *Conf* constructed with operators $\_,\_,\_ : Expression\ Substitution\ Store \rightarrow Conf$ and $\_,\_,\_ : Predicate\ Substitution\ Store \rightarrow Conf$ in $\mathcal{U}$ are precisely the rewrites that arise from conditions of the equations in $\mathcal{C}$.

$$\begin{array}{rl} \mathcal{U} \vdash \mathtt{v} := op(E_1, \ldots, E_n), sto & \rightarrow \\ op(E_1, \ldots, E_n), \mathtt{v} := E, sto & \rightarrow \\ op(E'_1, \ldots, E_n), \mathtt{v} := E, sto & \rightarrow^* \\ op(m_1, \ldots, m_n), \mathtt{v} := E, sto & \rightarrow^* \\ m, \mathtt{v} := E, sto & \rightarrow \\ \mathtt{skip}, update(sto, \mathtt{v}, m) & \end{array}$$

$$\mathcal{C} \vdash \frac{op(E_1, \ldots, E_n) \rightarrow^* op(m_1, \ldots, m_n) \wedge op(m_1, \ldots, m_n) \rightarrow^* m}{\mathtt{v} := op(E_1, \ldots, E_n), sto \rightarrow \mathtt{skip}, update(sto, \mathtt{v}, m)}$$

□

Due to space constraints, we do not present the complete Maude code. It can be obtained in full at https://github.com/ChristianoBraga/gsl-semantics. In what follows, module `GSL-SYNTAX` implements the syntax of GSL and module `GSL-SEMANTICS` implements the unconditional rewriting logic semantics in Fig. 3. We present a subset of the replacement rules. Following Section 3, the equations are annotated with the `variant` attribute. Moreover, the rule for unbounded choice is annotated as `nonexec` as it can not be executed by rewriting. It is specified in a logic programming style and requires narrowing search to be animated. We specify unbounded choice for both natural numbers and Boolean values.

```
(fmod GSL-SYNTAX is
  inc GSL-EXPRESSION .
  inc GSL-PREDICATE .
  sort Substitution UChoice PlaceHolder .
  subsort Variable < Expression Predicate .
  subsort UChoice < Substitution .
  --- Simple substitution
  op _:=_ : Variable Expression -> Substitution [ctor] .
  --- "Does nothing" substitution
  op skip : -> Substitution [ctor] .
  --- Pre-condition substitution
  op _|_ : Predicate Substitution -> Substitution [ctor] .
  --- Bounded choice substitution
  op _'[']_ : Substitution Substitution -> Substitution [assoc comm ctor] .
  --- Guarded substitution
  op _==>_ : Predicate Substitution -> Substitution [ctor] .
```

```
   --- Unbounded choice substitution
   op @_._ : PlaceHolder Substitution -> UChoice [ctor] .
endfm)
(mod GSL-SEMANTICS is
   inc GSL-SYNTAX .
   inc GSL-STORE .
   sort Conf .
   op abort : -> Conf .
   op _`,_ : Substitution Store ~> Conf .
   op _`,_`,_ : Predicate Substitution Store ~> Conf [frozen] .
   op _`,_`,_ : Expression Substitution Store ~> Conf [frozen] .
   op _`[_/_`] : Expression GNat PlaceHolder -> Expression .
   op _`[_/_`] : Expression GBool PlaceHolder -> Expression .
   op _`[_/_`] : Predicate GNat PlaceHolder -> Predicate .
   op _`[_/_`] : Predicate GBool PlaceHolder -> Predicate .
   op _`[_/_`] : Substitution GNat PlaceHolder -> Substitution .
   op _`[_/_`] : Substitution GBool PlaceHolder -> Substitution .
   var V V1 V2 : Variable . vars E E1 E2 : Expression . var STO STO' : Store . vars G G1 G2 : GNat .
   var P : Predicate . vars S S1 S2 : Substitution . var PH : PlaceHolder . var B B1 B2 : GBool .
   eq [simple1] : ((V := E) , STO) = (E, (V := E) , STO) [variant] .
   eq [simple2] : (V1, (V2 := E) , ((V1 |-> G) STO)) = (G, (V2 := E) , ((V1 |-> G) STO)) [variant] .
   eq [simple2] : (V1 + E1, (V2 := E2) , ((V1 |-> G) STO)) = (G + E1, (V2 := E2) , ((V1 |-> G) STO)) [variant] .
   eq [simple2] : (G1, (V := E) , ((V |-> G2) STO)) = (skip , ((V |-> G1) STO)) [variant] .
   eq [pre1] : (P | S , STO) = (P , P | S , STO) [variant] .
   eq [pre2] : (V1 eq E, (P | S) , ((V1 |-> G) STO)) = (G eq E, (P | S) , ((V1 |-> G) STO)) [variant] .
   eq [pre3] : (true, P | S , STO) = (S , STO) [variant] .
   eq [pre4] : (false, P | S , STO) = abort [variant] .
   rl [bchoice] : S1 [] S2 => S1 .
   eq [guard1] : (P ==> S , STO) = (P , (P ==> S) , STO) [variant] .
   eq [guard2] : ((V1 eq E), (P ==> S) , ((V1 |-> G) STO)) = ((G eq E), (P ==> S) , ((V1 |-> G) STO)) [variant] .
   eq [guard3] : (true, (P ==> S) , STO) = (S , STO) [variant] .
   eq [guard4] : (false, (P ==> S) , STO) = (skip, STO) [variant] .
   rl [uchoice] : (@ PH . S), STO => (S [ G / PH ]), STO [nonexec] .
   rl [uchoice] : (@ PH . S), STO => (S [ B / PH ]), STO [nonexec] .
   --- Equations for _[_/_]
   --- We assume that alpha-renaming has been performed.
   eq [repl-simple] : (V := E) [ G / PH ] = V := ( E [ G / PH ] ) .
   eq [repl-simple] : (V := E) [ B / PH ] = V := ( E [ B / PH ] ) .
   eq [repl-guarded-subst] : (P ==> S) [ G / PH ] = ((P [ G / PH ]) ==> (S [ G / PH])) .
   eq [repl-guarded-subst] : (P ==> S) [ B / PH ] = ((P [ B / PH ]) ==> (S [ B / PH])) .
   eq [repl-ph-gnat] : PH [ G / PH ] = G .
   eq [repl-ph-gbool] : PH [ B / PH ] = B .
   eq [repl-and] : (PH and P) [ B / PH ] = B and (P [ B / PH]) .
   eq [repl-ph-sum] : (PH + E) [ G / PH ] = G + (E [ G / PH ]) .
endm)
```

Example 2. In this example we illustrate how B machine specifications can be animated using narrowing. We include a new module `SIMPLIFIED-ABSTRACT-MACHINE-NOTATION` (for a subset of B machine commands) defining equations that specify how some commands of the Abstract Machine Notation (AMN) can be understood as syntactic sugar [1, pp. 266–267] to GSL commands. The semantics of `CHOICE` is given in terms of bounded choice and `ANY` in terms of unbounded choice, for instance.

```
(mod SIMPLIFIED-ABSTRACT-MACHINE-NOTATION is
      inc GSL-SEMANTICS .
   sort NeSubstitutionSet .
   subsort Substitution < NeSubstitutionSet .
   op CHOICE_END : NeSubstitutionSet -> Substitution .
   op ANY_WHERE_THEN_END : PlaceHolder Predicate Substitution -> Substitution .
   eq CHOICE SS END = SS .
   eq S OR T = S [] T .
   eq S OR (T OR SS) = S [] (T OR SS) .
   eq ANY X WHERE P THEN S END = (@ X . (P ==> S)) .
endm)
```

Now, if we execute the very simple AMN program below, devised only to exercise the symbolic part of GSL semantics, where x is place holder and y a variable,

```
CHOICE
ANY x WHERE true THEN y := x + s(0)END OR
ANY x WHERE true THEN y := x END
END
```

Maude produces a solution where y is either bound to some value (that is, x) or the successor of some value.

```
1 search [,2] in SAMN−EXAMPLE : CHOICE ANY x WHERE true THEN y := x + s(0)END OR ANY x WHERE true
      THEN y := x END END,STO:Store ˜>∗
2 skip, STO':Store .
3 Solution 1
4 STO':Store −−> #13:Store y |−> #7:GNat ;
5 STO:Store −−> #13:Store y |−> #12:GNat
6 Solution 2
7 STO':Store −−> #13:Store y |−> s(#7:GNat);
8 STO:Store −−> #13:Store y |−> #12:GNat
9 No more solutions.
```

5 Final Remarks

In this paper we propose a Rewriting Logic semantics for the Generalized Substitution Language, a normal form for B specifications in the B method. This semantics is symbolically executable in the Maude Rewriting Logic system.

Before we conclude, a final remark on the semantics of the *uchoice* rule. We could have used explicit metavariables (e.g. [11]) to specify the semantics of GSL unbounded choice. However, the use of unbounded variables on the right hand-side of a rule is a simple and sound specification technique, from a logical stand-point. With symbolic execution, such as narrowing, we need not mix representation levels, that is, the meta-level and the object level in the semantics. One should also note that, in practice, only the unconditional semantics exists. The conditional one exists only at the theoretical level to bridge the gap between the operational semantics for GSL and the executable one in Maude.

Future work includes supporting the complete syntax of AMN, the inclusion of additional automated analysis techniques and a full account of refinement.

References

1. J.-R. Abrial. *The B-book - Assigning Programs to Meanings.* Cambridge Univ. Press, 2005.
2. K. Bae and J. Meseguer. The Linear Temporal Logic of Rewriting Maude Model Checker. In *Rewriting Logic and Its Applications*, volume 6381 of *LNCS*, pages 208–225, 2010.
3. P. Chartier. *B'98: Recent Advances in the Development and Use of the B Method: Second International B Conference Montpellier, France, April 22–24, 1998 Proceedings*, chapter Formalisation of B in Isabelle/HOL, pages 66–82. Springer Berlin Heidelberg, 1998.
4. M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. M. Oliet, J. Meseguer, and C. Talcott. *Maude Manual*. SRI International, July 2016. Version 2.7.1.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

6. D. Déharbe and S. Merz. *Formal Aspects of Component Software: 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, chapter Software Component Design with the B Method — A Formalization in Isabelle/HOL, pages 31–47. Springer International Publishing, Cham, 2016.
7. S. Dunne. *ZB 2002:Formal Specification and Development in Z and B: 2nd International Conference of B and Z Users Grenoble, France, January 23–25, 2002 Proceedings*, chapter A Theory of Generalised Substitutions, pages 270–290. Springer Berlin Heidelberg, 2002.
8. N. Martí-Oliet and J. Meseguer. *Handbook of Philosophical Logic*, chapter Rewriting Logic as a Logical and Semantic Framework, pages 1–87. Springer Netherlands, 2002.
9. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73 – 155, 1992.
10. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
11. A. Verdejo and N. Martí-Oliet. *Implementing CCS in Maude*, pages 351–366. Springer US, Boston, MA, 2000.
12. F. Zeyda, B. Stoddart, and S. Dunne. *ZB 2005: Formal Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005. Proceedings*, chapter A Prospective-Value Semantics for the GSL, pages 187–202. Springer Berlin Heidelberg, 2005.

Automatic generation of focused proof systems

Elaine Pimentel[1]⋆ and Björn Lellmann[2]⋆⋆

[1] Department of Mathematics, UFRN, Brazil
[2] Department of Computer Languages, TU Wien, Austria

Abstract. In this work we show how to automatically generate focused sequent systems. For that, we start by capturing the object-level behavior in linear logic (LL) and use the well established LL meta-level reasoning in order to determine which object level inference rules are invertible or not. From that, we can propose a focused version of the non-invertible object-level rules based on the polarity of the encoded clause in LL.

1 Introduction

In a series of works [3,2,7,4], it has been shown how to use linear logic as a framework for specifying various proof systems. This not only allowed using the rich and well stablished linear logic meta-theory in order to reason about these systems, but also provided automatic proof search to all of them.

For example, by a simple analysis on the shape of rules, one can determine whether the specified system has cut-elimination, or when the system admits initial axioms restricted to atoms.

There are some other logical features that have never been analysed, though. One of them is invertibility of the object inference rules: what the encodings can tell about it? This analysis is important since one can use the notion of invertibility in order to produce a *focused* version of the specified logic, hence providing a better system for doing proof search.

The goal of this work is to generate focused systems automatically from sequent systems specified in linear logic. As a side effect, we can determine when an object-level inference rule is invertible or not.

Although not all sequent systems can be specified in linear logic, there is a great number of representative systems that can be adequately captured. Some of those still lack a good notion of focusing.

Also, establishing a general algorithm for proving invertibility of rules is important enough: like cut-elimination, this is the kind of property that has to be proven by hand, and in a case-by-case analysis. Automatizing this process seems to be a great step forward, and it can be used in different logical frameworks.

The rest of the paper is organized as follows: Section 2 presents the process of transforming sequent systems into LL clauses; Section 3 brings to the surface some new and not yet explored question of invertibility of inference rules of object-level sequent

⋆ Funded by CNPq and CAPES.
⋆⋆ Funded by the EU under Marie Skłodowska-Curie grant agreement No. 660047.

systems; Section 4 presents the inverse side of the encodings, revealing the behavior of object logics behind the LL encodings, and proposing focused systems from them and giving some working examples; finally, Section 5 concludes the paper, with some exciting future directions to be pursued.

2 From sequent systems to linear logic

In this section we show how to specify sequent systems in linear logic. We quickly present focused linear logic.

2.1 Linear logic and focusing

The connectives of linear logic can be divided into two classes: the *negative* $⅋$, $\bot$, $\&$, $\top$, $\forall$, ?; and the *positive* $\oplus$, 0, $\otimes$, 1, $\exists$, !. Observe that one class is the de Morgan dual of the other. A formula is *positive* if it is a negated atom or its top-level logical connective is positive. Similarly, a formula is *negative* if it is an atom or its top-level logical connective is negative. Atoms can be given any polarity. A *literal* is either an atomic formula or a negated atomic formula.

Polarities on formulas determine the so called phases of proof construction: in the *negative* phase of proof construction, no backtracking on the selection of inference rules is necessary, while in the *positive* phase choices within inference rules can lead to failures for which one may need to backtrack. This means that focused proofs can be seen (bottom-up) as a sequence of alternations between negative and positive phases.

The one-sided version of the focused proof system LLF is given in Figure 1.

2.2 Bipoles

The bipoles form a very special class of formulae, since they are totally decomposed in one focused step.

Definition 1. *A* monopole *formula is a linear logic formula that is built up from atoms and occurrences of the negative connectives, with the restriction that* ? *has atomic scope. A* bipole *is a positive formula built from monopoles and negated atoms using only positive connectives, with the additional restriction that* ! *can only be applied to a monopole.*

Using the linear logic distributive properties, monopoles are equivalent to formulas of the form

$$\forall x_1 \dots \forall x_p [\&_{i=1,\dots,n} ⅋_{j=1,\dots,m_i} B_{i,j}],$$

where the $B_{i,j}$ are either atoms or the result of applying ? to an atomic formula. Similarly, bipoles can be rewritten as formulas of the form

$$\exists x_1 \dots \exists x_p [\oplus_{i=1,\dots,n} \otimes_{j=1,\dots,m_i} C_{i,j}],$$

where $C_{i,j}$ are either negated atoms, monopole formulas, or the result of applying ! to a monopole formula. Notice that the units $\top$, 0, $\bot$, and 1 are 0-ary versions of $\&$, $\oplus$, $⅋$,

Negative rules

$$\frac{\Psi;\Delta \Uparrow L}{\Psi;\Delta \Uparrow \bot, L}\,[\bot] \qquad \frac{\Psi;\Delta \Uparrow F, G, L}{\Psi;\Delta \Uparrow F \parr G, L}\,[\parr] \qquad \frac{\Psi, F;\Delta \Uparrow L}{\Psi;\Delta \Uparrow ?F, L}\,[?]$$

$$\frac{}{\Psi;\Delta \Uparrow \top, L}\,[\top] \qquad \frac{\Psi;\Delta \Uparrow F, L \quad \Psi;\Delta \Uparrow G, L}{\Psi;\Delta \Uparrow F \,\&\, G, L}\,[\&] \qquad \frac{\Psi;\Delta \Uparrow F[y/x], L}{\Psi;\Delta \Uparrow \forall x.F, L}\,[\forall]$$

Positive rules

$$\frac{}{\Psi;\cdot \Downarrow 1}\,[1] \qquad \frac{\Psi;\Delta_1 \Downarrow F \quad \Psi;\Delta_2 \Downarrow G}{\Psi;\Delta_1, \Delta_2 \Downarrow F \otimes G}\,[\otimes] \qquad \frac{\Psi;\cdot \Uparrow F}{\Psi;\cdot \Downarrow\, !F}\,[!]$$

$$\frac{\Psi;\Delta \Downarrow F_1}{\Psi;\Delta \Downarrow F_1 \oplus F_2}\,[\oplus_l] \qquad \frac{\Psi;\Delta \Downarrow F_2}{\Psi;\Delta \Downarrow F_1 \oplus F_2}\,[\oplus_r] \qquad \frac{\Psi;\Delta \Downarrow F[t/x]}{\Psi;\Delta \Downarrow \exists x.F}\,[\exists]$$

Identity, Decide, and Reaction rules

$$\frac{}{\Psi;A \Downarrow A^\perp}\,[I_1] \qquad \frac{}{\Psi, A;\cdot \Downarrow A^\perp}\,[I_2] \qquad \frac{\Psi;\Delta \Downarrow F}{\Psi;\Delta, F \Uparrow \cdot}\,[D_1] \qquad \frac{\Psi, F;\Delta \Downarrow F}{\Psi, F;\Delta \Uparrow \cdot}\,[D_2]$$

In $[I_1]$ and $[I_2]$, A is atomic; in $[D_1]$ and $[D_2]$, F is not an atom.

$$\frac{\Psi;\Delta, F \Uparrow L}{\Psi;\Delta \Uparrow F, L}\,[R\Uparrow] \qquad \text{provided that } F \text{ is positive or an atom}$$

$$\frac{\Psi;\Delta \Uparrow F}{\Psi;\Delta \Downarrow F}\,[R\Downarrow] \qquad \text{provided that } F \text{ is negative}$$

Fig. 1. Focused proof search in linear logic LLF. The variable y in the $[\forall]$ rule is restricted so that it is not free in any formula of its conclusion.

and $\otimes$, respectively. Given this normal representation of bipoles and according to the focusing discipline, it turns out that, once introduced, a bipole is completely decomposed into its atomic subformulas, a fact illustrated by the following bipole derivation.

$$\dfrac{\cdots \quad \dfrac{\dfrac{\dfrac{\Psi';\Gamma' \Uparrow \cdot}{\Psi;\Gamma' \Uparrow \parr_{j=1,\ldots,m_i} ?A_{i,j}}\,[\parr,?] \quad \cdots}{\Psi;\Gamma' \Uparrow \forall x_1 \ldots \forall x_p[\&_{i=1,\ldots,n} \parr_{j=1,\ldots,m_i} ?A_{i,j}]}\,[\forall,\&]}{\Psi;\Gamma' \Downarrow\, ! \forall x_1 \ldots \forall x_p[\&_{i=1,\ldots,n} \parr_{j=1,\ldots,m_i} ?A_{i,j}]}\,[!] \quad \cdots}{\Psi;\Gamma \Downarrow \exists x_1 \ldots \exists x_t[\oplus_{i=1,\ldots,k} \otimes_{j=1,\ldots,q_i} C_{i,j}]}\,[\exists,\oplus,\otimes]$$

Here $A_{i,j}$ is atomic for all i, j. If the connective ! is not present, then the rule ! is replaced by the rule $R\Downarrow$.

2.3 Specifying logical systems

Bipoles play an important role when specifying inference rules. In fact, since they are decomposed entirely and at once, one can guarantee that the application of an object level rule is matched by focusing on bipole a clause: no other meta-level action can be done in the middle of the process.

Specifying sequents Let *obj* be the type of object-level formulae and let $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ be two meta-level predicates of type $obj \to o$. Object-level sequents of the form $B_1, \ldots, B_n \vdash C_1, \ldots, C_m$ (where $n, m \geq 0$) are specified as the multiset $\lfloor B_1\rfloor, \ldots, \lfloor B_n\rfloor, \lceil C_1\rceil, \ldots, \lceil C_m\rceil$ within the LLF proof system. The $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ predicates identify which object-level formulas appear on which side of the sequent – brackets down for left and brackets up for right. We will assume that atoms in LLF are given a negative bias.

Specifying inference rules Inference rules are specified by a re-writing clause that replaces the active formulae in the conclusion by the active formulae in the premises. The linear logic connectives indicate how these object level formulae are connected: contexts are copied (&) or split (⊗), in different inference rules (⊕) or in the same sequent (⅋). As an example, the additive and multiplicative versions of the right inference rules for conjunction are, respectively

$$\frac{\Delta \vdash \Gamma, A \quad \Delta \vdash \Gamma, B}{\Delta \vdash \Gamma, A \wedge B}\ \wedge_{RA} \qquad \frac{\Delta_1 \vdash \Gamma_1, A \qquad \Delta_2 \vdash \Gamma_2, B}{\Delta_1, \Delta_2 \vdash \Gamma_1, \Gamma_2, A \wedge B}\ \wedge_{RM}$$

These inference rules can be specified in linear logic using the clauses

$$(\wedge RA) \quad \exists A, B(\lceil A \wedge B\rceil^{\perp} \otimes (\lceil A\rceil \,\&\, \lceil B\rceil)). \qquad (\wedge RM) \quad \exists A, B(\lceil A \wedge B\rceil^{\perp} \otimes (\lceil A\rceil \otimes \lceil B\rceil)).$$

Specifying cut, initial and structural rules We shall assume that all systems have the exchange rule (that is, we will not deal with non-commutative logics). Weakening and contraction are handled by a direct use of linear logic exponentials and the structural clauses

$$\text{(Neg)} \quad \exists B(\lceil B\rceil^{\perp} \otimes ?\lceil B\rceil) \qquad\qquad \text{(Pos)} \quad \exists B(\lfloor B\rfloor^{\perp} \otimes ?\lfloor B\rfloor).$$

The initial rule, which asserts that the sequent $B \vdash B$ is provable, can be specified by the clause

$$(\mathit{Init}) \qquad \exists B(\lfloor B\rfloor^{\perp} \otimes \lceil B\rceil^{\perp}).$$

The multiplicative cut rule can be specified by the clause

$$(\mathit{Cut}) \qquad \exists B(\lceil B\rceil \otimes \lfloor B\rfloor),$$

The *Init* and *Cut* clauses together prove that $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ are duals of each other, that is, $\forall B(\lfloor B\rfloor^{\perp} \equiv \lceil B\rceil)$.

When specifying a system (logical, computational, etc) into a meta level framework, it is mandatory that the specification is *faithful*, that is, one step of computation on the object level should correspond to one step of logical reasoning in the meta level. This is what is called *adequacy* [5].

Definition 2. *A specification of an object sequent system is* proof-adequate *if provability is preserved by the specification. If the adequacy can be shown for (open) derivations (such as inference rules themselves), then we call the specification* adequate.

Adequate specifications can be used for automatic proof search since each focused phase in the LLF derivation corresponds to the application of a logical object rule and vice versa. However, in this work we will not focus our attention on proof search, but how the meta-level theory obtained from adequate specifications can be better analyzed so to produce better object logical systems.

3 Canonical systems and invertibility

The following definition captures the notion of introduction rules in the meta-level approach, using bipoles.

Definition 3. *Let $\diamond$ be an object-level connective of arity n ($n \geq 0$) and $q \in \{\lfloor\cdot\rfloor, \lceil\cdot\rceil\}$. A clause of the shape*

$$\exists\overline{A}\exists\overline{x}.[h(\diamond, \overline{A}) \otimes B]$$

is called an introduction clause *if it is a closed bipole formula where:*

- *$h(\diamond, \overline{A})$ is a positive formula of the form $q(\diamond(\overline{A}))^{\perp}$.*
- *An atom occurring in B is of the form $q(A)$ or $q(A(z))$ with $A \in \overline{A}$ and $z \notin \{\overline{x}\}$.*

We call $h(\diamond, \overline{A})$ the head *and B the* body *of the introduction clause.*

Definition 4. *A* canonical clause *is an introduction clause restricted so that, for every pair of atoms of the form $\lfloor T \rfloor$ and $\lceil S \rceil$ in a body, the head variable of T differs from the head variable of S. A* canonical proof system theory *is a set $\mathcal{X}$ of bipoles such that (i) the* Init *and* Cut *clauses are members of $\mathcal{X}$, (ii) structural clauses (*Pos *and* Neg*) may be members of $\mathcal{X}$, and (iii) all other clauses in $\mathcal{X}$ are canonical (introduction) clauses.*

We will now analyse better the meaning, at the meta level, of invertibility at the object level.

Definition 5. *An inference rule is* invertible *if, the derivability of its premisses follows from the derivability of its conclusion.*

It is easy to see that in LL, for example, all the connectives classified as negatives have invertible rules. In general, for proving that a rule is invertible one has to analyse two cases: when the conclusion formula is principal in the derivation to not. If it is principal with the application of an introduction rule for the main connective, then the result is trivial, not requiring induction. Also, if the conclusion formula is not principal the result follows trivially by the inductive hypothesis. Difficulties may arise when the formula is principal either in an instance of contraction or the initial axiom, as the following example shows.

Example 1. Consider a logical system containing only the following rules

$$\frac{}{\Gamma, A \vdash \Delta, A}\ init \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \star B \vdash \Delta}\ \star_L \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \star B}\ \star_R$$

The left and right introduction rules for $\star$ look like the rules for the additive disjunction, apart from the fact that $\star$ has only one right rule. It is well known that the left rule for additive disjunction is invertible, in general. But the rule $\star_L$ *is not* invertible. In fact, if it was, from any proof of $\Gamma, A \star B \vdash \Delta$ there would be a proof of $\Gamma, A \vdash \Delta$ and a proof of $\Gamma, B \vdash \Delta$. But, although $A \star B \vdash A \star B$ is provable, $B \vdash A \star B$ *does not have* a proof.

Hence, the invertibility of a rule depends on all the rules of the system. The problem with the system above is that the initial axiom cannot be restricted to atoms.

Definition 6. *Let $\mathcal{X}$ be a canonical proof system and $\diamond$ an object-level connective of arity $n \geq 0$. Furthermore, let the formulas*

$$\exists\overline{A}\exists\overline{x}.[h_l(\diamond,\overline{A}) \otimes B_l] \quad \textit{and} \quad \exists\overline{A}\exists\overline{x}.[h_r(\diamond,\overline{A}) \otimes B_r]$$

be the left and right introduction clauses for $\diamond$. The object-level connective $\diamond$ is coherent *if B_l and B_r are dual LL formulas, that is, if the sequents*

$$Cut, Init; \cdot \Uparrow \forall\overline{x}.(B_l)^{\perp} \equiv B_r \quad \textit{and} \quad Cut, Init; \cdot \Uparrow \forall\overline{x}.(B_r)^{\perp} \equiv B_l$$

are provable in LLF. A canonical system is called coherent if all object-level connectives have coherent introduction rules.

The following theorem relates coherent systems, cut elimination and initial axioms (see [4] for the proof).

Theorem 1. *If a canonical proof system $\mathcal{X}$ is coherent then it has cut-elimination and the initial axiom can be restricted to atoms.*

The following result is now given by the invertibility of right rules for negative connectives in linear logic.

Theorem 2. *Let $\mathcal{X}$ be a coherent system corresponding to an adequate specification of an object logical system in LLF and let $\exists\overline{A}\exists\overline{x}.[h(\diamond,\overline{A}) \otimes B]$ be the introduction clause for an object level connective $\diamond$. Then B is a monopole if and only if the corresponding object logical rule is invertible.*

Proof. For one direction, suppose that $h(\diamond,\overline{A}) = (\lceil\diamond(\overline{A})\rceil)^{\perp}$ and let π be a proof of $\Gamma \vdash \Delta, \diamond(\overline{A})$. If the last rule applied in π is $\diamond_R$, then the result follows trivially. Otherwise, the proof proceeds by induction on the size of π.

Suppose that the last rule applied in π is the initial axiom on $\diamond(\overline{A})$, which means that $\diamond(\overline{A}) \in \Gamma$. Hence π corresponds to a LLF proof focusing on the init clause, instantiated by $\diamond(\overline{A})$. Since the system is coherent, this proof can be replaced by a meta-level proof where init is restricted to atomic object formulas. The last rule of such a proof has to necessarily be a focusing over the $(\diamond_R)$ clause. In fact, since B_r is negative, it will be totally decomposed in its subformulas, and applying such a clause won't change provability of the meta level sequent.

Since the specification is adequate, such a meta-level proof can be faithfully translated into an object level proof π', which last applied rule is $\diamond_R$.

If the last rule applied in π is the initial axiom on some other formula, then $\diamond(\overline{A})$ will be weakened and it can be substituted by any other formula, including the ones in the premises of $\diamond_R$.

Finally, if $\diamond(\overline{A})$ is not principal in the last rule applied in π, then the result holds easily, using the inductive hypothesis.

The other direction is trivial since invertibility is inherited from object to meta-level by adequate specifications. □

Example 2. Consider the introduction rules for $\star$, presented in Example 1. As we noted, the left rule for $\star$ *is not* invertible, although it is encoded by the LL clause

$$\lfloor A \star B\rfloor^{\perp} \otimes \lfloor A\rfloor \,\&\, \lfloor B\rfloor$$

where the body is a monopole. But the system is not canonical since, in the presence of *Cut* and *Initial*,

$$(\lfloor A \rfloor \,\&\, \lfloor B \rfloor)^{\perp} \equiv \lceil A \rceil \oplus \lceil B \rceil \not\equiv \lceil A \rceil$$

That is, the hypothesis of the theorem does not hold.

4 From linear logic to focused systems

We will now show how to recover formulas, contexts and sequents from specified systems. Then we show an algorithm for transforming introduction clauses into focused rules.

Formulas. The set $U(F)$ of underlying object level formulas $\underline{F}$ of a meta-level formula F is defined recursively as

- if $F = \lceil A \rceil$ then $\underline{F} = A$;
- if $F = \lfloor A \rfloor$then $\underline{F} = (A)^{\perp}$;
- $U(F) = \{\underline{F_i} \mid F_i \text{ is an atomic subformula of } F\}$.

Contexts. We will denote by Γ, Δ object contexts.

Sequents. We will denote by $\Gamma \to [A], \Delta$ a focused sequent over a formula on the right, $\Gamma, [A] \to \Delta$ a focused sequent over a focused formula on the left and $\Gamma \Rightarrow \Delta$ an unfocused sequent. Given $\underline{F_i} \in U(F)$, if $\underline{F_i} = (A)^{\perp}$ then A is placed in the antecedent of a sequent. Otherwise, $\underline{F_i}$ is placed in the succedent.

Rules. An introduction clause of the form $\exists\overline{A}\exists\overline{x}.\lceil \diamond(\overline{A}) \rceil^{\perp} \otimes B$ corresponds to a focused object rule

$$\frac{\left\{\Gamma_j \Updownarrow \Delta_j\right\}_j}{\Gamma, \Updownarrow x : \diamond(\overline{A}), \Delta}\ [\diamond_R]$$

where formulas in Γ_j, Δ_j are in $\Gamma, \Delta, U(B)$ and the number of premises as well as the shape of a context (i.e. if it is split or copied) is entirely determined by the meta-level specification; moreover, $\Updownarrow \in \{\Rightarrow, \to\}$ is completely determined by the polarity of the body of the clause as follows:

1. the conclusion sequent is focused on $\diamond(\overline{A})$ if and only if the body is positive; if the body is negative, then the premise sequents are unfocused;
2. one of the premises sequent is focused if and only if B is a positive formula with no banged subformulas or no negative non-atomic subformulas;
3. if B is positive with a negative non-atomic subformula, then the correspondent premise sequent is unfocused.

The algorithm for left rules follows the same lines as above.

Theorem 3. *Let OS be an object system and* ***OS*** *and adequate encoding of it. Then the focused system obtained is correct and complete w.r.t OLS.*

Proof. Direct from the adequacy of the encoding. □

$$
\begin{array}{llll}
(\supset L) & \lfloor A \supset B \rfloor^{\perp} \otimes (\lceil A \rceil \otimes \lfloor B \rfloor). & (\supset R) & \lceil A \supset B \rceil^{\perp} \otimes (\lfloor A \rfloor \parr \lceil B \rceil). \\
(\wedge L) & \lfloor A \wedge B \rfloor^{\perp} \otimes (\lfloor A \rfloor \oplus \lfloor B \rfloor). & (\wedge R) & \lceil A \wedge B \rceil^{\perp} \otimes (\lceil A \rceil \,\&\, \lceil B \rceil). \\
(\vee L) & \lfloor A \vee B \rfloor^{\perp} \otimes (\lfloor A \rfloor \,\&\, \lfloor B \rfloor). & (\vee R) & \lceil A \vee B \rceil^{\perp} \otimes (\lceil A \rceil \oplus \lceil B \rceil). \\
(\forall_c L) & \lfloor \forall_c B \rfloor^{\perp} \otimes \exists x \lfloor Bx \rfloor. & (\forall_c R) & \lceil \forall_c B \rceil^{\perp} \otimes \forall x \lceil Bx \rceil. \\
(\exists_c L) & \lfloor \exists_c B \rfloor^{\perp} \otimes \forall x \lfloor Bx \rfloor. & (\exists_c R) & \lceil \exists_c B \rceil^{\perp} \otimes \exists x \lceil Bx \rceil. \\
(f_c L) & \lfloor f_c \rfloor^{\perp} \otimes \top. & (t_c R) & \lceil t_c \rceil^{\perp} \otimes \top.
\end{array}
$$

Fig. 2. Specification of the classical logic's system **LK**.

Fig. 3. Focused version of **LK**.

$$
\frac{\Gamma_1 \to \Delta_1, [A] \quad \Gamma_2, [B] \to \Delta_2}{\Gamma_1, \Gamma_2, [A \supset B] \to \Delta_1, \Delta_2} \supset L \qquad \frac{\Gamma, A \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \supset B} \supset R
$$

$$
\frac{\Gamma, [A] \to \Delta}{\Gamma, [A \wedge B] \to \Delta} \wedge L_1 \qquad \frac{\Gamma, [B] \to \Delta}{\Gamma, [A \wedge B] \to \Delta} \wedge L_2 \qquad \frac{\Gamma \Rightarrow \Delta, A \quad \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \wedge B} \wedge R
$$

$$
\frac{\Gamma, A \Rightarrow \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \vee B \Rightarrow \Delta} \vee L \qquad \frac{\Gamma \to \Delta, [A]}{\Gamma \to \Delta, [A \vee B]} \vee R_1 \qquad \frac{\Gamma \to \Delta, [B]}{\Gamma \to \Delta, [A \vee B]} \vee R_2
$$

$$
\frac{\Gamma, B[t/x] \to \Delta}{\Gamma, \forall_c.B \to \Delta} \forall L \qquad \frac{\Gamma \Rightarrow \Delta, B[y/x]}{\Gamma \Rightarrow \Delta, \forall_c.B} \forall R
$$

$$
\frac{\Gamma, B[y/x] \Rightarrow \Delta}{\Gamma, \exists_c.B \Rightarrow \Delta} \exists L \qquad \frac{\Gamma \to \Delta, B[t/x]}{\Gamma \to \Delta, \exists_c.B} \exists R
$$

$$
\frac{}{\Gamma, f_c \Rightarrow \Delta} f_c L \qquad \frac{}{\Gamma \Rightarrow \Delta, t_c} \exists R
$$

4.1 Some relevant examples

In Figure 2 we present the adequate encoding of the system LK, for classical logic. We proved in [4] that this specification is adequate, and that the specified system is coherent.

Hence, by Theorem 2, the rules $\Rightarrow R$, $\wedge R$, $\forall_c R$, $f_c L$, $f_c R$, $\vee L$ and $\exists_c L$ are invertible and the resulting focused system is given in Figure 3.

It is straightforward to do the same reasoning for clauses in Figures 4 and 5.

It is interesting to note that not all sequent systems can be adequately encoded in LL. In fact, in [6] we showed that well known sequent modal systems (e.g for K and S4) could not be adequately encoded in LL. In that work, we presented two solutions: either change the sequent system (for using, e.g. labelled systems) or change the meta-level system (use the so called *subexponentials*).

It is pretty straightforward to change our approach in order to handle labelled systems. In fact, it is sufficient adding *relational formulas*. Since this is also enough for describing extensions of sequent systems based in *nested sequents*, all we have described so far should work for more involving proof systems. This is an ongoing work. See Figure 6 for the encoding in LL of the modal system KD, together with its automatically generated focused system.

$$
\begin{array}{llll}
(\supset L) & \lfloor A \supset B \rfloor^{\perp} \otimes (\lceil A \rceil \otimes \lfloor B \rfloor). & (\supset R) & \lceil A \supset B \rceil^{\perp} \otimes (\lfloor A \rfloor \parr \lceil B \rceil). \\
(\cap L) & \lfloor A \cap B \rfloor^{\perp} \otimes (\lfloor A \rfloor \oplus \lfloor B \rfloor). & (\cap R) & \lceil A \cap B \rceil^{\perp} \otimes (\lceil A \rceil \,\&\, \lceil B \rceil). \\
(\cup L) & \lfloor A \cup B \rfloor^{\perp} \otimes (\lfloor A \rfloor \,\&\, \lfloor B \rfloor). & (\cup R) & \lceil A \cup B \rceil^{\perp} \otimes (\lceil A \rceil \oplus \lceil B \rceil). \\
(\forall_i L) & \lfloor \forall_i B \rfloor^{\perp} \otimes \exists x \lfloor Bx \rfloor. & (\forall_i R) & \lceil \forall_i B \rceil^{\perp} \otimes \forall x \lceil Bx \rceil. \\
(\exists_i L) & \lfloor \exists_i B \rfloor^{\perp} \otimes \forall x \lfloor Bx \rfloor. & (\exists_i R) & \lceil \exists_i B \rceil^{\perp} \otimes \exists x \lceil Bx \rceil.
\end{array}
$$

Fig. 4. Specification **LM** minimal logic

$$
\begin{array}{llll}
(\multimap L) & \lfloor A \multimap B \rfloor^{\perp} \otimes (\lceil A \rceil \otimes \lfloor B \rfloor). & (\multimap R) & \lceil A \multimap B \rceil^{\perp} \otimes (\lfloor A \rfloor \parr \lceil B \rceil). \\
(\otimes L) & \lfloor A \otimes B \rfloor^{\perp} \otimes (\lfloor A \rfloor \parr \lfloor B \rfloor). & (\otimes R) & \lceil A \otimes B \rceil^{\perp} \otimes (\lceil A \rceil \otimes \lceil B \rceil). \\
(\&L_1) & \lfloor A \,\&\, B \rfloor^{\perp} \otimes \lfloor A \rfloor. & (\&R) & \lceil A \,\&\, B \rceil^{\perp} \otimes (\lceil A \rceil \,\&\, \lceil B \rceil). \\
(\&L_2) & \lfloor A \,\&\, B \rfloor^{\perp} \otimes \lfloor B \rfloor. & (\oplus R_1) & \lceil A \oplus B \rceil^{\perp} \otimes \lceil A \rceil. \\
(\oplus L) & \lfloor A \oplus B \rfloor^{\perp} \otimes (\lfloor A \rfloor \,\&\, \lfloor B \rfloor). & (\oplus R_2) & \lceil A \oplus B \rceil^{\perp} \otimes \lceil B \rceil. \\
(\parr L) & \lfloor A \parr B \rfloor^{\perp} \otimes (\lfloor A \rfloor \otimes \lfloor B \rfloor). & (\parr R) & \lceil A \parr B \rceil^{\perp} \otimes (\lceil A \rceil \parr \lceil B \rceil). \\
(!\,L) & \lfloor !\,B \rfloor^{\perp} \otimes ?\lfloor B \rfloor. & (!\,R) & \lceil !\,B \rceil^{\perp} \otimes !\lceil B \rceil. \\
(?L) & \lfloor ?B \rfloor^{\perp} \otimes\ !\lfloor B \rfloor. & (?R) & \lceil ?B \rceil^{\perp} \otimes ?\lceil B \rceil. \\
(\forall_l L) & \lfloor \forall_l B \rfloor^{\perp} \otimes \exists x \lfloor Bx \rfloor. & (\forall_l R) & \lceil \forall_l B \rceil \otimes \forall x \lceil Bx \rceil. \\
(\exists_l L) & \lfloor \exists_l B \rfloor^{\perp} \otimes \forall x \lfloor Bx \rfloor. & (\exists_l R) & \lceil \exists_l B \rceil^{\perp} \otimes \exists x \lceil Bx \rceil. \\
(1L) & \lfloor 1 \rfloor^{\perp} \otimes\ \perp. & (1R) & \lceil 1 \rceil^{\perp} \otimes\ !\top. \\
(\perp\ L) & \lfloor \perp \rfloor^{\perp} \otimes\ !\top. & (\perp\ R) & \lceil \perp \rceil^{\perp} \otimes\ \perp. \\
(0L) & \lfloor 0 \rfloor^{\perp} \otimes \top. & (\top R) & \lceil \top \rceil^{\perp} \otimes \top.
\end{array}
$$

Fig. 5. Specification of object-level linear logic **LL**.

5 Concluding remarks and future work

In this work, we have moved forward into the direction of providing focused sequent systems for given object level systems. In a nutshell, what we did was to establish a kind of Galois connection between the object level and the meta level, in this case, linear logic (LL).

The idea is to take an object level sequent system, adequately encode it into LL, verify that is has *good properties* and come back to the object level via a focused system. Observe that provability is guaranteed in the process, while we end up with less proofs at the end of the process. That is, our connection refines the proof space, leaving only the normal proofs.

A criticism one may have in this method is that it should be easier to get the focused system directly from the original one, without passing through LL. Although this is true in essence, focusing was first defined for LL and all focused proof systems proposed so far relies somehow in this base notion, either via translations or via semantic graphs. There is no "general" notion of what focusing is.

Here we propose that focusing is, in fact, inherited by LL, and we adequately translate this to the object level.

There are a number of ways of continuing this work, we will present two: (1) as said in Section 4.1, it is easy to extend this work to labelled systems, hence being able to handle *modal logics*; and (2) it is still open the cut-elimination criteria to extensions of sequent systems, like linear nested systems [1]. With that, one could think of automatically

$$\frac{}{\mathcal{R}, X, x : A \Rightarrow x : A, Y}\ \mathsf{init} \qquad \frac{\mathcal{R}, X, x : A, x : B \to Y}{\mathcal{R}, X, x : A \wedge B \to Y}\ \wedge_L \qquad \frac{\mathcal{R}, X \to x : A, Y \quad \mathcal{R}, X \to x : B, Y}{\mathcal{R}, X \to x : A \wedge B, Y}\ \wedge_R$$

$$\frac{xRy, X \to Y, y : A \quad zRx \to zRx}{zRx, X \Rightarrow Y, [x : \Box A]}\ \Box_R \qquad \frac{xRy, y : A \to Y \quad xRy \to xRy}{xRy, [x : \Box A] \Rightarrow Y}\ \Box_L$$

$$\frac{xRy, X, y : A \to Y \quad zRx \to zRx}{zRx, [x : \Box A], X \Rightarrow Y}\ \mathsf{d}$$

Fig. 6. Some rules of the focused labeled-linear sequent calculus $\mathsf{FLLS}_{\mathsf{KD}}$ for KD .

generating focused systems for a gamma of systems not adequately represented by sequent calculi.

References

1. Lellmann, B., Pimentel, E.: Proof search in nested sequent calculi. In: Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. pp. 558–574 (2015)
2. Miller, D., Pimentel, E.: Using linear logic to reason about sequent systems. In: Egly, U., Fermüller, C.G. (eds.) International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. LNCS, vol. 2381, pp. 2–23. Springer (2002)
3. Miller, D., Pimentel, E.: Linear logic as a framework for specifying sequent calculus. In: van Eijck, J., van Oostrom, V., Visser, A. (eds.) Logic Colloquium '99: Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic, pp. 111–135. Lecture Notes in Logic, A K Peters Ltd (2004)
4. Miller, D., Pimentel, E.: A formal framework for specifying sequent calculus proof systems. Theor. Comput. Sci. 474, 98–116 (2013)
5. Nigam, V., Miller, D.: A framework for proof systems. J. of Automated Reasoning 45(2), 157–188 (2010)
6. Nigam, V., Pimentel, E., Reis, G.: An extended framework for specifying and reasoning about proof systems. J. Log. Comput. 26(2), 539–576 (2016), `http://dx.doi.org/10.1093/logcom/exu029`
7. Pimentel, E., Miller, D.: On the specification of sequent systems. In: LPAR 2005: 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. pp. 352–366. No. 3835 in LNAI (2005)

Towards Simpler Theorem-Proving of Graph Grammars with Negative Application Conditions

Guilherme Azzi and Leila Ribeiro

Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
Email: {ggazzi, leila}@inf.ufrgs.br

Abstract. Graph Grammars are a rule-based computational model, describing the states of a system as graphs and its behavior as rewriting rules. In order to ensure that a graph grammar has the desired properties, theorem proving is a promising approach: it can guarantee that a specification is respected, without being affected by the combinatorial explosion of the state-space generated even by simple grammars. Applying the existing approaches for theorem-proving graph grammars can however be very complex. Within an ongoing effort to ease an existing approach, this paper proposes a simple modification, in an attempt to simplify the proofs. The impact of this modification is discussed, as well as some remaining roadblocks to the practical application of the approach.

1 Introduction

The formalism of Graph Grammars [12,5] is a rule-based computational model, describing the states of a system as graphs and its behavior as rewriting rules. It is an inherently data-driven model, with a visual and intuitive representation, while still possessing solid formal underpinnings. The rule-based, data-driven and non-deterministic nature of Graph Grammars makes them well-suited to model reactive, concurrent and distributed systems. They are also a good match for model-based software development, where the transformation of visual models is central to the process.

A rich theory of graph rewriting was developed in the last few decades, providing several extensions to the original formalism, which are often conveninent or even necessary when modelling nontrivial systems. A particularly common extension is the use of negative application conditions [7] (NACs), which extend rules with patterns that forbid their application if present. Another vital extension is the use of attributed nodes and edges, that is, associating values of certain data types (e.g. numbers) to them, as well as allowing rules to contain variables and terms.

Besides the aforementioned extensions, several verification techniques were developed to ensure that a graph grammar has the desired properties. Among them, theorem proving is attractive despite the high effort involved in the proofs. It can guarantee that a grammar has certain properties, which static analysis

techniques such as critical pair analysis and concurrent rules [5,10,9] cannot achieve in general. Furthermore, unlike model checking, it is not affected by the large or infinite state-spaces that are generated even by simple grammars.

An approach for theorem proving graph grammars was previously investigated in [11,3], where Graph Grammars were encoded in event-B [1]. Invariants of the grammar were then formulated in first-order logic with set theory and proved in the Rodin theorem prover [2]. It was noted, however, that "it might be a very complex task to state the desired property as well as develop the proof, even for developers with a strong theoretical background" [3], which led to an ongoing effort to improve the usability of this framework. An alternative encoding of NACs was proposed [3], and specification patterns [4] as well as proof strategies [8] were proposed.

This papers follows this ongoing effort, proposing an alternative encoding of graph grammars into event-B with the aim of simplifying proofs of invariants. An example graph grammar is encoded and verified, and the difficulty of verification is compared to the original encoding. During the verification process, a few major roadbloacks to make this approach practicable were identified and are described in this paper.

The remainder of this paper is organized as follows. Sections 2 and 3 briefly present the definitions of graph grammars with NACs and the formalism of event-B, respectively. Section 4 proposes a modification to the translation of graph grammars to event-B. Section 5 discusses the impact of the changes on the development of proofs, listing the main difficulties that still remain, while section 6 summarizes and concludes this paper.

2 Graph Grammars with NACs

In this section we briefly review the main definitions of graph grammars with NACs from a set-theoretical perspective, rather than the more common categorial approach. These definitions are presented in more detail in [5,3].

A ***graph*** G is a tuple (V_G, E_G, src_G, tgt_G), where V_G is a set of nodes, E_G is a set of edges, and $src_G, tgt_G : E_G \to V_G$ assigns the sources and targets, respectively, to the edges. A ***graph morphism*** $f : G \to H$ is a pair of functions $(f_V : V_G \to V_H, f_E : E_G \to E_H)$, respecting sources and targets, i.e.

$$\forall e \in E_G \cdot f_V(src_G(e)) = src_H(f_E(e)) \land f_V(tgt_G(e)) = tgt_H(f_E(e))$$

A ***partial graph morphism*** is a graph morphism where functions over nodes and edges are partial.

Real applications handle entities of different types, which may relate to each other in several different ways. An example from [3] is the specification of a token ring protocol, where all nodes are of the same type, but different edges are used to indicate the ownership of a token or message, the next node in the ring, and which node is currently active. This may be supported by taking a particular graph as the ***type graph***. Figure 1 shows the type graph for the token ring example.

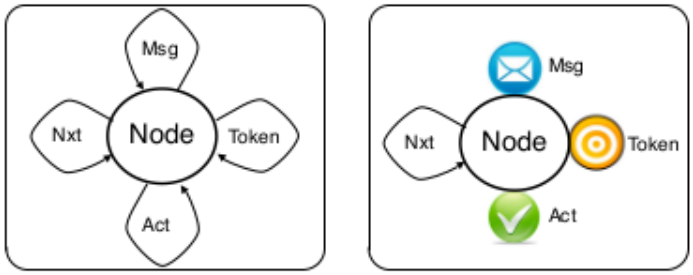


Fig. 1. Types for the token ring protocol, from [3]

A ***typed graph*** is then a tuple $G^T = (G, tG, T)$, where G is the graph itself, T is the type graph and $tG : G \to T$ assigns the types to nodes and edges of G. A ***typed graph morphism from*** G^T ***to*** H^T is just a graph morphism $f : G \to H$ that also preserves types, i.e.

$$\forall v \in V_G \cdot tG_V(v) = tH_V(f_V(v))$$

$$\forall e \in E_G \cdot tG_E(e) = tH_E(f_E(e))$$

In order to model the dynamic aspects of the system, we need transformations rules. Such rules should assert that, whenever a given pattern is found on a graph, certain nodes/edges should be deleted and other nodes/edges should be created. A rule may thus be modeled by an injective T-typed partial graph morphism $\rho : L \rightarrowtail R$. The left-hand side L defines the pattern that should be found in the graph. Any nodes/edges that are not mapped by ρ will be deleted, and the nodes/edges of the right-hand side R that are outside the image of ρ will be created.

Besides using a pattern to permit the application of a rule (the left-hand side), it is often useful to define patterns that prohibit the application. Thus, we define a negative application condition with a graph N expressing the forbidden pattern, as well as a morphism $nac : L \to N$, embedding the left-hand side into the forbidden pattern. The complete definition of a ***rule with NACs*** is thus a pair (ρ, NAC), where $\rho : L \rightarrowtail R$ is the rule morphism and $NAC = \{nac_i : L \to N_i\}_i$ is a (finite) set of NACs.

As an example, two rules from [3] may be seen in Figure 2, describing part of a token ring protocol. These two rules specify what a node that possesses the token may do, as long as it is not yet active: it may either become active and send a message (Rule 1), or just hand the token to the next node (Rule 2).

When specifying a complete system, we must give its rules as well as an initial state. Thus, a ***Graph Grammar with NACs*** is a tuple (T, G_0, R), where T is a type graph, G_0 is the initial T-typed graph and R is a set of rules with NACs.

In order to apply a rule to an instance graph G, we need a ***match*** $m : L \to G$ embedding the left-hand side into the instance graph. In order for a rule to be

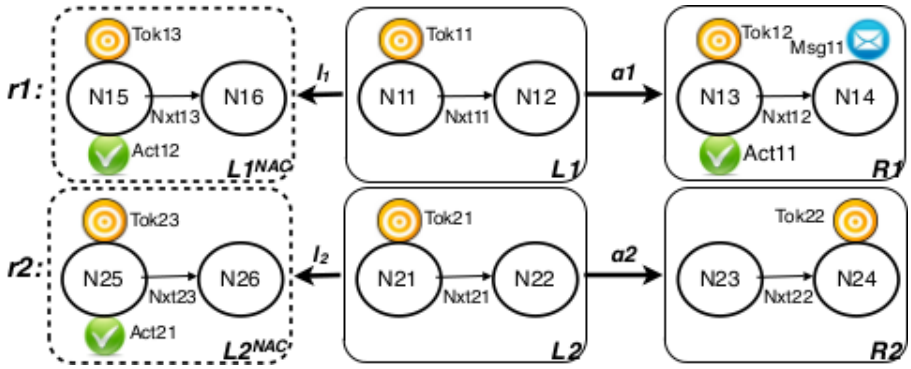


Fig. 2. Some rules for a token ring protocol, from [3]

applicable to a particular match, both the NACs and some *gluing conditions* must be respected: the *dangling condition* states that no edges may be incident to deleted nodes, except those that will be deleted by the rule; the *identification condition* requires that the match may not map two nodes or edges of the left-hand side into the same element of the instance graphs, if one of them is deleted and the other, preserved. While some approaches don't require the dangling condition, implicitly deleting any non-matched edges incident to a deleted node, the identification condition is generally considered useful.

The formal definition of the gluing conditions, of NAC satisfaction by a match, and of the rewriting are omitted due to lack of space. They may be found in [6].

3 Event-B

Event-B [1] is a state-based formalism for modelling, specification and verification of systems. It is based in first-order logic and set theory. An event-B model is composed of a static part, called the context, and a dynamic part, called the machine. A context (c, s, A) contains sets c and s of constants and set names, respectively, as well as a collection of axioms $A(c, s)$ describing them. A machine $(v, I, init, E)$ contains a set of variables v that describe its state, a set of invariants $I(v)$ that should be respected in every (reachable) state, an initialization predicate $init(v')$ that constrains the initial state of the machine, and a set E of events, which describe the execution of the machine. An event (G, BA) is defined by a guard $G(v)$, which restricts the states in which the event may occur, and a before-after predicate $BA(v, v')$, which describes the modified state with respect to the original one. When multiple events are enabled in a state, the semantics of event-B executes one of them non-deterministically. A formal definition of the event-B semantics is omitted due to lack of space.

The Rodin Plaftorm [2] provides extensive tool suppport for modeling with event-B. It automatically generates the proof obligations that are necessary to

prove that the invariants hold in every reachable state, and assists in proving such obligations.

4 Modified Translation of Graph Grammars to Event-B

In [3], graph grammars are translated to event-B by a very literal representation of their parts, which is put into the context. Graphs are represented as sets of nodes and edges, as well as source and target functions. (Partial) morphisms between graphs are represented as pairs of (partial) functions, one over nodes and another over edges. Rules are represented by their left- and right-hand graphs and the partial morphism between them, as well as objects and morphisms for the NACs. The application of each rule is then defined as an event of the machine, postulating the existence of the appropriate morphisms and defining the modified graph in terms of the sets and functions.

Example 1. The translation of Rule 1 from Figure 2 to event-B will generate the following event:

```
event r1
  any mV // node-component of the match
      mE // edge-component of the match
      newMsg11 newAct11 // fresh names for the created edges
  where
    // Match is total on nodes and edges
    @grd_mV  mV ∈ VertL1 → VertG
    @grd_mE  mE ∈ EdgeL1 → EdgeG

    // Match respects types, source and target
    @grd_tV      ∀v · v ∈ VertL1 ⇒ tL1V(v) = tGV(mV(v))
    @grd_tE      ∀e · e ∈ EdgeL1 ⇒ tL1E(e) = tGE(mE(e))
    @grd_srctgt  ∀e · e ∈ EdgeL1 ⇒ mV(sourceL1(e)) = srcG(mE(e)) ∧
                                   mV(targetL1(e)) = tgtG(mE(e))

    // The created edges are distinct and new
    @grd_new_newMsg11   newMsg11 ∈ ℕ \ EdgeG
    @grd_new_newAct11   newAct11 ∈ ℕ \ EdgeG
    @grd_diff_newEdges  newMsg11 ≠ newAct11

    // The NAC is satisfied
    @grd_NAC1 ¬(∃nV, nE ·
                  nV ∈ VertNAC1 ⤖ VertG  ∧  nE ∈ EdgeNAC1 ⤖ EdgeG ∧
                  (∀v · v ∈ VertNAC1 ⇒ tNAC1V(v) = tGV(nV(v))) ∧
                  (∀e · e ∈ EdgeNAC1 ⇒ tNAC1E(e) = tGE(nE(e))) ∧
                  (∀e · e ∈ EdgeNAC1 ⇒ nV(sourceNAC1(e)) = srcG(nE(e)) ∧
                                       nV(targetNAC1(e)) = tgtG(nE(e))) ∧
                  nV ∘ nac1V = mV  ∧  nE ∘ nac1E = mE)
  end
  then
    @act_E    EdgeG := EdgeG ∪ {newAct11, newMsg11}
    @act_src  srcG := srcG ∪ {newAct11 ↦ mV(N11), newMsg11 ↦ mV(N12)}
    @act_tgt  tgtG := tgtG ∪ {newAct11 ↦ mV(N11), newMsg11 ↦ mV(N12)}
    @act_tE   tGE := tGE ∪ {newAct11 ↦ Act, newMsg11 ↦ Msg}
  end
end
```

Despite this translation being correct, it was noted that properties dealing with concrete elements of the state were very hard to prove [3], since the non-existence of the concrete forbidden elements must be manually proven from the

non-existence of particular morphisms. The authors then proposed an alternative, set-theoretic formulation of NACs, postulating the non-existence of the forbidden nodes and edges explicitly. This alternative formulation was proven equivalent to the original one, and was successfully used to prove the desired properties.

Example 2. The alternative translation of the NAC in Rule 1 of Figure 2 will generate the following guard:

```
// The NAC is satisfied
@grd_NAC1 ¬(∃forbAct12.
                forbAct12 ⊆ EdgeG \ mE[EdgeL1] ∧ tGE(forbAct12) = Act ∧
                srcG(forbAct12) = mV(N11) ∧ tgtG(forbAct12) = mV(N11))
```

Although some amount of indirection was removed by the alternate NAC translation, some indirection still remains when reasoning about the match. We therefore a similar approach: instead of describing the match as a function, the events should contain variables for the individual nodes and edges of the match. An informal description of the encoding follows below.

for $v \in V_L$ **do**
  add new variable $\mathtt{v}_v$ to the parameters of the event;
  add the guard $\left[\mathtt{v}_v \in \mathtt{Edge_G} \wedge \mathtt{tG_V}(\mathtt{v}_v) = \mathtt{type}_{tL_V(v)}\right]$;
end
for $e \in E_L$ **do**
  add new variable $\mathtt{e}_e$ to the parameters of the event;
  add the guard $\left[\begin{array}{l}\mathtt{e}_e \in \mathtt{Edge_G} \wedge \mathtt{tG_E}(\mathtt{e}_e) = \mathtt{type}_{tL_E(e)} \wedge \\ \mathtt{src_G}(\mathtt{e}_e) = \mathtt{v}_{src_L(e)} \wedge \mathtt{tgt_G}(\mathtt{e}_e) = \mathtt{v}_{tgt_L(e)}\end{array}\right]$;
end
add the variables $\mathtt{Match_V}$, $\mathtt{Match_E}$, $\mathtt{Del_V}$ and $\mathtt{Del_E}$ to the parameters;
add a guard $\left[\mathtt{Match_V} \subseteq \mathtt{Vert_G} \wedge \mathtt{Match_V} = \left\{\mathtt{v}_{v_1}, \mathtt{v}_{v_2}, \dots\right\}\right]$, $v_i \in V_G$;
add a guard $\left[\mathtt{Del_V} \subseteq \mathtt{Match_V} \wedge \mathtt{Del_V} = \left\{\mathtt{v}_{v_1}, \mathtt{v}_{v_2}, \dots\right\}\right]$, $v_i \in V_G \setminus dom(\rho_V)$;
add similar guards for $\mathtt{Match_E}$ and $\mathtt{Del_E}$;
add guards stating that the gluing conditions and NACs are satisfied;
for $v \in V_R \setminus rng(\rho_V)$ **do**
  add new variable $\mathtt{newVert}_v$;
  add a guard $[\mathtt{newVert}_v \in \mathbb{N} \setminus \mathtt{Vert_V}]$;
end
for $e \in E_R \setminus rng(\rho_E)$ **do**
  add new variable $\mathtt{newEdge}_e$;
  add a guard $[\mathtt{newEdge}_e \in \mathbb{N} \setminus \mathtt{Edge_V}]$;
end
add variables $\mathtt{New_V}$ and $\mathtt{New_E}$ to the parameters;
add a guard $\left[\mathtt{New_V} = \left\{\mathtt{newVert}_{v_1}, \mathtt{newVert}_{v_2}, \dots\right\}\right]$, $v_i \in V_R \setminus rng(\rho_V)$;
add a guard $\left[\mathtt{partition}(New_V, \left\{\mathtt{newVert}_{v_1}\right\}, \left\{\mathtt{newVert}_{v_2}\right\}, \dots)\right]$, $v_i \in V_R \setminus rng(\rho_V)$;
add a theorem $[\mathtt{Vert_G} \cap \mathtt{NewV} = \emptyset]$ to the guards;
add similar guards defining $\mathtt{New_E}$;
add variables $\mathtt{tNew_V}$, $\mathtt{tNew_E}$, $\mathtt{srcNew}$ and $\mathtt{tgtNew}$ to the parameters;
add guards defining the types of the functions $\mathtt{tNew_V}$, $\mathtt{tNew_E}$, $\mathtt{source_{New}}$ and $\mathtt{target_{New}}$;
add a guard $\left[\mathtt{tNew_V} = \left\{\mathtt{v}_{v_1} \mapsto \mathtt{type}_{tR_V(v_1)}, \mathtt{v}_{v_2} \mapsto \mathtt{type}_{tR_V(v_2)}, \dots\right\}\right]$, $v_i \in V_R \setminus rng(\rho_V)$;
add a guard $\left[\mathtt{tNew_E} = \left\{\mathtt{e}_{e_1} \mapsto \mathtt{type}_{tR_E(e_1)}, \mathtt{e}_{e_2} \mapsto \mathtt{type}_{tR_E(e_2)}, \dots\right\}\right]$, $e_i \in E_R \setminus rng(\rho_E)$;
add a guard $\left[\mathtt{src_{New}} = \left\{\mathtt{e}_{e_1} \mapsto \mathtt{v}_{src_R(e_1)}, \mathtt{e}_{e_2} \mapsto \mathtt{v}_{src_R(e_2)}, \dots\right\}\right]$, $e_i \in E_R \setminus rng(\rho_E)$;
add a guard $\left[\mathtt{tgt_{New}} = \left\{\mathtt{e}_{e_1} \mapsto \mathtt{v}_{tgt_R(e_1)}, \mathtt{e}_{e_2} \mapsto \mathtt{v}_{tgt_R(e_2)}, \dots\right\}\right]$, $e_i \in E_R \setminus rng(\rho_E)$;
add actions similar to the original translation;

Algorithm 1: Modified encoding of Graph Grammars in event-B

Example 3. The alternative translation of Rule 1 from Figure 2 would generate the following event:

```
event r1
  any n1 n2 // Matched nodes
      nxt tok // Matched edges
      newMsg newAct // Created edges
      MatchV MatchE DelE NewE // Sets of nodes and edges
      tNewE sourceNew targetNew // Updates to functions
  where
    // Nodes and edges have correct types, sources and targets
    @type_n1   n1 ∈ VertG ∧ tGV(n1) = Node
    @type_n2   n2 ∈ VertG ∧ tGV(n2) = Node
    @type_nxt  nxt ∈ EdgeG ∧ tGE(nxt) = Nxt ∧ srcG(nxt) = n1 ∧ tgtG(nxt) = n2
    @type_tok  tok ∈ EdgeG ∧ tGE(tok) = Tok ∧ srcG(tok) = n1 ∧ tgtG(tok) = n1

    // Definition of the matched and deleted elements
    @def_MatchV MatchV = {n1, n2}
    @def_MatchE MatchE = {nxt, tok}
    @def_DelE   DelE ⊆ EdgeG ∧ DelE = ∅

    // Created nodes/edges are distinct and new
    @type_newMsg   newMsg ∈ ℕ \ EdgeG
    @type_newAct   newAct ∈ ℕ \ EdgeG
    @def_NewE      NewE ⊆ ℕ ∧ NewE = {newAct, newMsg}
    @grd_diff_NewE partition(NewE, {newAct}, {newMsg})
    theorem @grd_newDisjointEdgeG EdgeG ∩ NewE = ∅

    // The NACs are satisfied
    @grd_NAC1 ¬(∃forbAct · forbAct ∈ EdgeG \ MatchE ∧
                tGE(forbAct) = Act ∧
                srcG(forbAct) = n1 ∧ tgtG(forbAct) = n1)

    // Types, sources and targets of created elements
    @type_tNew_E tNewE ∈ NewE → EdgeT
    @type_sourceNew sourceNew ∈ NewE → (VertG \ DelV) ∪ NewV
    @type_targetNew targetNew ∈ NewE → (VertG \ DelV) ∪ NewV

    @def_tNew_E tNewE = {newAct ↦ Act, newMsg ↦ Msg}
    @def_sourceNew sourceNew = {newAct ↦ n1, newMsg ↦ n2}
    @def_targetNew targetNew = {newAct ↦ n1, newMsg ↦ n2}
  end
  then
    @act_E   EdgeG := EdgeG ∪ NewE
    @act_src srcG := srcG ∪ sourceNew
    @act_tgt tgtG := tgtG ∪ targetNew
    @act_tE  tGE := tGE ∪ tNewE
  end
end
```

A formal definition of this translation is left for future work, as well as proof of its correctness with respect to graph rewriting. Essentially, graph morphisms that make up rules are no longer explicitly encoded as pairs of functions. Instead, they are implicitly encoded in the definitions of the events.

5 Discussion and Open Problems

A complete graph grammar for the token ring protocol was encoded with the modified translation, comprising 6 events and the initialization. Eight invariants were proven to hold for the system: there is always a single token edge; there is at most one active node, and any active node also holds the token; edges of

types Tok and Act always have the same node as source and target; the instance graph has a finite amount of nodes and edges, and in particular a finite amount of edges of type Tok and Act. These are exactly the invariants proven in [3].

In order to prove that the invariants hold, Rodin generated a total of 135 proof obligations (POs). Of these, 102 were proven automatically by the tool, leaving only 33 to be proven manually. These were divided in the same two categories as [3]: POs that could be proven only by clicking on symbols and using Rodin's provers are classified as **easy**, while POs that required manually instantiating or introducing hypotheses are classified as **manual**. The POs are further divided into **translation POs**, which guarantee the type-correctness of the generated model, and **application POs** which are related to the properties specified by the user. Table 1 compares the number of POs for the original and modified encodings.

In general, the difficulty of the proofs remained unchanged: the invariants that were proven automatically almost always remained so, and the harder proofs remained complex. The exception was proving that rules which modify the set of nodes preserve the correct types for the source and target functions. The number of assumptions available for each proof, however, has increased dramatically, which might negatively impact the performance of automatic theorem provers, especially with larger problems.

The number of application POs was reduced because two of the invariants were specified as theorems, since they follow from the other invariants. Thus, they generated only 2 POs, instead of 2 per rule. The increase in translation POs is due to the increased number of guards, whose well-definedness needed to be proven, which was done automatically by the tool.

Table 1. Proof obligations for the token ring example.

Encoding	**Translation POs**				**Application POs**			
	POs	Auto	Easy	Manual	POs	Auto	Easy	Manual
Original	64	60	1	0	56	14	18	24
Modified	91	87	0	4	44	15	5	24

The modified encoding had little impact in the difficulty of proving invariants. A possible downside is the increase in assumptions that are available during any given proof, which might hinder the performance of automatic provers. It may be argued, on the other hand, that the models generated by the modified encoding are easier to understand.

During the verification process, a few major difficulties were identified. These are described below.

– *Proving that elements of a particular type are not changed is nontrivial*, especially when the invariant involves multiple elements of non-changed types, despite it being intutively obvious. This is also a very common kind of proof,

since there is typically only a few rules that modifies elements of any given type.
- *Proving that the types are preserved by rule application is nontrivial.* The type function before and after application are syntactically very different, and proving that a preserved element has the same type is nontrivial. This is, however, necessary for any property that quantifies over elements of a particular type.
- *Recurring patterns are hard to abstract.* Despite many of the proofs being very similar, small differences such as the names of variables make the copying of proofs in Rodin fail. The difficulty of abstracting these patterns as lemmas and/or tactics results in a lot of repeated work.

6 Conclusions and Future Work

Within an ongoing effort to ease the approach of [11,3] for verifying properties of graph grammars with NACs by theorem proving, this paper proposed a simple modification. The encoding of rules in event-B was changed, being written in terms of the concrete nodes and edges of the match instead of using graph morphisms. Although the difficulty of the verification remained relatively unchanged, the resulting model is arguably easier to understand.

During the verification of a graph grammar, some major roadblocks to the application of this approach were identified. These could direct future research in theorem proving graph grammars.

Currently, we are working on a formal definition of the modified translation, as well as a proof of its correctness with respect to graph transformation. As future work we also intend on adapting the modified translation to handle attributed graphs, and on developing a theory for typed graphs in the Theory Plug-in of the Rodin Platform to further ease the proofs. Another interesting direction may be the encoding of graph grammars in other theorem provers such as Isabelle/HOL and Coq, although this translation would be more involved since the underlying logic doesn't have a native notion of event or transition.

Acknowledgements

The authors would like to acknowledge the brazilian agencies CNPq and CAPES for their support in the form of financial aid (VeriTes project/CNPq) and scholarships (CAPES).

References

1. Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

3. Simone A. C. Cavalheiro, Luciana Foss, and Leila Ribeiro. Theorem proving graph grammars with attributes and negative application conditions. Submitted to Theoretical Computer Science.
4. Simone A. C. Cavalheiro, Luciana Foss, and Leila Ribeiro. Specification patterns for properties over reachable states of graph grammars. In *Formal Methods: Foundations and Applications: 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings*, pages 83–98. Springer, 2012.
5. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
6. Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation: Part ii: Single pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars*, pages 247–312, 1997.
7. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3, 4):287–313, 1996.
8. Luiz C. L. Junior, Simone A. C. Cavalheiro, and Luciana Foss. Theorem proving graph grammars: Strategies for discharging proof obligations. In *Formal Methods: Foundations and Applications: 16th Brazilian Symposium, SBMF 2013, Brasilia, Brazil, September 29 - October 4, 2013, Proceedings*, pages 147–162. Springer, 2013.
9. Leen Lambers. *Certifying rule-based models using graph transformation.* PhD thesis, Berlin Institute of Technology, 2009.
10. Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict detection for graph transformation with negative application conditions. In *Graph Transformations: Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2006.
11. Leila Ribeiro, Fernando L. Dotti, Simone A. da Costa, and Fabiane C. Dillenburg. Towards theorem proving graph grammars using event-b. *Electronic Communications of the EASST*, 30, 2010.
12. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

Abordagens Metodológicas para Ensino de Teoria da Computação, Linguagens Formais e Autômatos

Ícaro Andrade Souza[1] , Ecivaldo de Souza Matos[1] e Débora Abdalla Santos[1]

[1] Departamento de Ciência da Computação – Universidade Federal da Bahia (UFBA)
40170-110 – Salvador – BA – Brazil

{icaro.andrade; ecivaldo; abdalla}@ufba.br

Resumo. Teoria da Computação, Linguagens Formais e Autômatos, segundo as diretrizes curriculares nacionais (DCN), formam um conjunto de conteúdos fundamentais aos estudantes de cursos de graduação em Computação. Entretanto pesquisadores relatam a dificuldade de ensinar/aprender os conteúdos relacionados a essa área e buscam-se abordagens que amenizem os problemas vivenciados. Nesse sentido, este artigo apresenta um levantamento de estratégias de ensino de Teoria da Computação, Linguagens Formais e Autômatos, com objetivo de identificar abordagens didáticas em nível de graduação.
Palavras-chave: Teoria da Computação, Linguagens Formais e Autômatos, Educação em Computação, Metodologias de Ensino**.**

1 Introdução

Segundo as Diretrizes Curriculares Nacionais [7] em vigor, para todos os cursos de graduação em Computação (bacharelado/licenciatura), entre os conteúdos curriculares exigidos na formação tecnológica e básica estão Teoria da Computação (TC), Linguagens Formais e Autômatos (LFA).

Os conteúdos curriculares referentes à Teoria da Computação, Linguagens Formais e Autômatos referem-se aos fundamentos matemáticos da Computação, os quais analisam problemas que podem ser computados por um dado modelo teórico de Computação e, de uma forma geral, respondem quais são as capacidades e as limitações dos computadores [15] e [26].

Pesquisadores relatam que esses conteúdos são de suma importância para a formação acadêmica dos estudantes de Computação [8]. Outros ressaltam que conteúdos curriculares existentes em TC e LFA dão suporte a outras disciplinas, também fundamentais aos cursos de Computação, a exemplo de Compiladores e Inteligência Artificial [11] e [25].

Desde meados do século passado os conteúdos dessa área já começaram a estabelecer-se como uma importante fundamentação da Ciência da Computação [9]. Entretanto, segundo Chakraborty, professores e pesquisadores já percebiam que a TC e LFA possuem temas difíceis de ensinar e aprender.

Nesse sentido, Ezer e Trakhtenbrot [12] sinalizam que uma das habilidades-chave necessária aos estudantes de disciplinas referentes aos tópicos de TC e LFA é aptidão

para raciocínio matemático preciso, mas isso tem sido difícil e até mesmo assustador para os estudantes. Pirovani e Mataveli [21] argumentam que os estudantes consideram os tópicos dessas disciplinas áridos, abstratos, complexos e desvinculados de suas atividades profissionais, o que contribui para reduzir o interesse e a motivação pelo seu aprendizado. Borges [6] ressalta que o modo tradicional de ensino não consegue motivar os estudantes a se interessarem pela disciplina, entre outras razões, pois para esses estudantes não é clara a importância de certos conteúdos para a sua formação.

Aguiar e Oeiras [2], afirmam que ocorre "baixo uso de ferramentas computacionais de visualização destinadas a superar dificuldades de aprendizagem em Teoria da Computação", fato que, segundo eles, também pode contribuir para as dificuldades apresentadas por estudantes e professores. Outros pesquisadores, como Pirovani e Mataveli [21] declaram ainda que "professores têm dificuldades de encontrar formas alternativas para ensinar o conteúdo e tornar a disciplina de Teoria da Computação mais interessante para os alunos". O que, de acordo com esses autores, pode ser considerado como um ponto-chave ao desencadeamento de parte dos problemas existentes no ensino dos conteúdos de TC.

Tendo em vista os problemas relacionados ao ensino de conteúdos curriculares referentes à Teoria da Computação, Linguagens Formais e Autômatos, a pesquisa parcialmente relatada neste artigo visa identificar e analisar (i) recursos computacionais e (ii) metodologias educacionais para o ensino de conteúdos curriculares de Teoria da Computação, Linguagens Formais e Autômatos.

Neste artigo apresentamos os resultados parciais desta pesquisa: identificação e análise preliminar de metodologias educacionais para o ensino de Teoria da Computação, Linguagens Formais e Autômatos em cursos de graduação em Computação.

2 Metodologia

Este artigo relata o processo e os resultados da identificação e análise de metodologias/métodos de ensino de conteúdos curriculares de Teoria da Computação, Linguagens Formais e Autômatos, cuja investigação foi estruturada em duas fases: (i) levantamento de trabalhos sobre metodologias educacionais para o ensino de tópicos relacionados à TC e LFA e (ii) análise dos trabalhos mapeados.

A primeira fase correspondeu a uma pesquisa exploratória com objetivo de mapear os trabalhos relacionados a metodologias educacionais referentes à área; a segunda fase tratou da análise dos trabalhos encontrados na primeira fase, buscando identificar e analisar as abordagens existentes para o ensino de conteúdos relacionados a Teoria da Computação, Linguagens Formais e Autômatos.

Na pesquisa exploratória foram utilizados os seguintes termos de busca: ensino; Teoria da Computação; Linguagens Formais. Vale destacar que os termos foram buscados em português e em inglês. Foram consideradas seis fontes de dados: *Biblioteca Digital Brasileira de Teses e Dissertações (BDTD/IBICT); Biblioteca Digital Brasileira de Computação (BDBComp/SBC); Portal de Periódicos da Capes; Web of Science; ACM Digital Library; e a Revista Brasileira de Informática na Educação (RBIE)*. As bases foram selecionadas por indexarem a maioria dos veículos

qualificados na área de Ciência da Computação, bem como pela disponibilidade e acessibilidade.

Para o desenvolvimento da pesquisa exploratória foram definidos os seguintes critérios de seleção: trabalhos de pesquisa, desenvolvimento tecnológico e/ou relato de experiência publicados entre 2000 a 2015; escrito em português, inglês ou espanhol; acessíveis gratuitamente por pelo menos uma das fontes selecionadas; apresentar alguma abordagem que possibilite ensinar tópicos referentes aos conteúdos curriculares de Teoria da Computação, Linguagens Formais e Autômatos.

Buscando identificar e analisar metodologias/métodos de ensino da área, foram pesquisados 2.965 trabalhos e após sucessivos refinamentos foram selecionados 15 trabalhos, os quais apresentavam abordagens metodológicas para ensino de TC e LFA. Essa análise foi desenvolvida por meio da leitura completa dos trabalhos, identificando cada abordagem relatada nos textos e extraindo as seguintes informações: tipo de abordagem, conteúdos relacionados, ano de publicação e breve resumo descrevendo o método/metodologia.

3 Resultados

Nesta seção são apresentados os resultados parciais obtidos a partir da primeira e segunda fase do desenvolvimento da pesquisa. Esses resultados referem-se à identificação e análise preliminar dos trabalhos relacionados à área e das abordagens encontradas que possibilitam ensinar conteúdos curriculares de Teoria da Computação, Linguagens Formais e Autômatos.

3.1 Pesquisa Exploratória

A pesquisa exploratória resultou em 2.965 trabalhos referentes a recursos computacionais e/ou metodologias/métodos de ensino sobre a área. É importante salientar que foram utilizados, em algumas bases de dados, filtros secundários resultando em 2020 trabalhos para serem analisados.

Após a análise dos respectivos títulos, resumos e palavras-chave, 86 artigos foram considerados relevantes para a pesquisa. Entretanto, verificou-se que dentre os 86 artigos selecionados, alguns estavam duplicados, a partir daí foram eliminados os trabalhos ambíguos, restaram 63 artigos selecionados. Dentre os 63 trabalhos selecionados, 15 trabalhos descrevem abordagens metodológicas para ensino de tópicos referentes à TC e LFA e 48 apresentam recurso(s) computacional(is) desenvolvidos para auxiliar o ensino de conteúdos relacionados a TC e LFA.

3.2 Abordagens Metodológicas

Dentre as 15 produções encontrados (Tabela 1), três são estudos brasileiros disponíveis em português e 12 são artigos em inglês. Esses trabalhos foram classificados de acordo com os conteúdos abordados; a(s) teoria(s) ou princípio(s) de ensino/aprendizagem que são seguidos (abordagens teórico-metodológicas); e a utilização de recursos computacionais como ferramentas de treinamento e/ou

visualização. Essas abordagens serão apresentadas de acordo aos conteúdos abordados nas subseções abaixo.

Tabela 1. Abordagens metodológicas referentes ao ensino de Teoria da Computação, Linguagens Formais e Autômatos.

[1]Produção	[2]Conteúdo	[3]Teoria/Princípio	[4]Recursos Computacionais
Ruehr [22]	Autômatos, Árvores e DAG	-	JFLAP e LRR
Furtado [14]	Autômatos e Compiladores	-	-
Chesñevar; González e Maguitman [10]	TC	Teoria de aprendizagem construtivista e princípios da aprendizagem significativa	Minerva, Deus Ex Machina, JFLAP e Turing Machines
Sá e Bittencourt [23]	Autômatos	Avaliação, estruturação e solução de problemas	-
Verma [27]	Autômatos Finitos	-	JFLAP e LRR
Ezer e Trakhtenbrot [12]	Pumping Lemma	-	-
Ezer e Trakhtenbrot [13]	Expressões regulares	-	-
Sigman [25]	TC	Técnica de aprendizagem por descoberta e aprendizagem baseada em problemas	-
Arbe; Ortega e Conde [3]	Expressões regulares	-	-
Korte, et al. [17]	Autômatos	Abordagem construcionista	-
Ben-Ari [5]	Concorrência, Verificação e NDE	-	Jspin e SpinSpider
Armoni, Lewenstein e Ben-Ari [4]	NDE	-	-
Merceron [20]	Autômatos Finitos Determinísticos	-	-
Scarton e Aluisio [24]	TC	-	Learning objects e MERLOT3
Knobelsdorf, Kreitz e Böhne [16]	TC	Abordagem de aprendizagem cognitiva	-

[1] Referência do artigo que descreve a abordagem.
[2] Conteúdo abordado pelo método ou metodologia proposta.
[3] Teoria(s)/Princípio(s) cuja abordagem se baseia.
[4] Recursos computacionais que são utilizados pela abordagem.

3.2.1 Ensino de *Teoria da Computação*

Dentre as produções encontradas, quatro tratam especificamente do ensino da disciplina de Teoria da Computação. Buscam tornar a disciplina mais dinâmica, fazendo com que os estudantes se tornem mais ativos/participativos do processo de ensino/aprendizagem. A construção de cada uma dessas abordagens foi baseada em teoria/princípio de ensino/aprendizagem específicos.

A abordagem desenvolvida por Chesñevar, González e Maguitman [10], utiliza a combinação de diferentes estratégias de ensino a fim de tornar tópicos relacionados à Teoria da Computação mais interessantes e atrativos para os estudantes. Para tal essa metodologia utiliza a aprendizagem construtivista e alguns dos princípios da aprendizagem significativa. A abordagem propõe a utilização de diversos recursos (página web; *slides*; exercícios e recursos computacionais) para complementar o ensino tradicional e estimular a percepção dos estudantes da importância do conteúdo de Teoria da Computação na sua formação profissional.

Sigman [25] descreve a construção de um curso utilizando a técnica de aprendizagem por descoberta, conhecida como *Método de Moore* [29]. Essa técnica representa uma família de abordagens de aprendizagem baseada em problemas, que visam envolver de forma direta o estudante ao conteúdo estudado. As aulas são realizadas com uso de questões norteadoras permitindo que os estudantes descubram suas próprias capacidades para criar e aprender. Segundo o autor essa abordagem permitiu aos estudantes desenvolver as habilidades necessárias para encarar a matemática contida nos conteúdos relacionados à TC e LFA.

Outra abordagem nesse sentido tem o objetivo de tornar os alunos protagonistas dos processos de ensino e de aprendizagem, por meio da utilização de *learning objects e MERLOT3* [24]. O seu método incentiva os alunos a elaborarem conteúdos que estendam ou aprofundem os tópicos abordados em sala de aula, por fim o método propõe que os envolvidos apresentem o conteúdo produzido para a turma.

A última abordagem referente à TC descreve modificações pedagógicas em um curso de Teoria da Computação realizada na Universidade de Potsdam na Alemanha [16]. Tais modificações são motivadas por uma abordagem de "aprendizagem cognitiva", com objetivo de mostrar a aplicabilidade dos conteúdos e tornar mais fácil a aprendizagem. A proposta baseia-se em dividir a disciplina em três partes: aulas tradicionais, aulas tutoriais/palestras e aulas de exercícios. Tendo como diferencial as aulas tutoriais, em que os estudantes são incentivados a perguntar e discutir questões que ainda não estão claras após as sessões de aulas tradicionais.

3.2.2 Ensino de *Autômatos*

Seis abordagens metodológicas foram encontradas para ensino de Modelos Computacionais (Autômatos). Entretanto, duas delas tratam especificamente dos Autômatos Finitos, que são os reconhecedores do nível/tipo três da hierarquia de Chomsky. As demais abordagens tratam de toda a hierarquia, que é composta pelos: *Autômatos Finitos* (AF), *Autômatos de Pilha* (AP) e as *Máquinas de Turing* (MT).

Uma das abordagens específicas para Autômatos Finitos é a de Verman [27], que visa ilustrar a amplitude dos conceitos de AF através da integração das seguintes ferramentas de visualização: *Java Formal Languages and Automata Package*

(JFLAP) e *Laboratory for Rapid Rewriting* (LRR). Foi desenvolvido um material instrucional (conjunto de ferramentas pedagógicas e tecnológicas) que propõe, no primeiro momento, realizar uma revisão dos conceitos de Linguagens Formais e Autômatos, demonstrando suas aplicações e a variedade de modelos computacionais existentes. Em paralelo à revisão, o material sugere a introdução dos conceitos de computabilidade e conta com um conjunto de problemas que promove maior interação e experimentação dos estudantes com os conteúdos específicos.

A segunda abordagem é a proposta por Merceron [20], a qual relata a construção de um modelo para ensinar/aprender *Autômatos Finitos Determinísticos* (AFD). O autor relata que em sua instituição de ensino alguns estudantes não possuem experiência com programação, o que torna mais difícil motivá-los a estudar. Assim foi proposto um método com três passos para auxiliar os estudantes a projetarem AFD. O autor relata que o professor e o estudante serão guiados a resolver os modelos propostos segundo a ordem especificada e, assim, através de cada solução o aluno construirá novos conhecimentos.

No tocante às abordagens desenvolvidas para ensino dos Autômatos referentes à Hierarquia de Chosmky, Ruehr [22] propôs uma série de estratégias didáticas para tornar o curso mais agradável e interessante para os estudantes. O foco dessa abordagem é melhorar a aprendizagem, aumentando a visualização e a interação da disciplina, através das ferramentas JFLAP e LRR. Essa proposta busca ilustrar os modelos e conceitos discutidos durante as aulas, incluindo demonstrações de aplicações recentes dos conceitos referentes aos variados tipos de autômatos, incluindo também árvores e *Grafos Acíclicos Dirigidos* (DAG). Tal abordagem busca desenvolver aulas práticas em laboratório utilizando um conjunto de problemas com soluções.

A metodologia proposta por Furtado [14] refere-se à criação de uma nova disciplina para o ensino de Linguagens Formais e Autômatos, vinculada explicitamente ao ensino de Compiladores. Tal disciplina tem como objetivos principais apresentar uma visão geral do processo de compilação, sob o ponto de vista da implementação; e abordar conteúdos relacionados à Teoria da Computação, sob a ótica da Teoria das Linguagens Formais, enfatizando seus aspectos teóricos e suas aplicações. Com a abordagem proposta, o autor espera tornar os conteúdos relacionados a essa área mais agradáveis, aumentando o interesse dos estudantes pelos aspectos teóricos da computação e tornando o aprendizado mais efetivo.

Sá e Bittencourt [23] apresentam uma proposta curricular para a disciplina de Teoria da Computação. O objetivo geral dessa proposta é apresentar os principais modelos computacionais clássicos sob um ponto de vista unificado, baseado na avaliação, estruturação e solução de problemas. Para tal essa proposta visa estruturar o conteúdo programático para a disciplina de Teoria da Computação, buscando motivar o estudo de conceitos ligados à computação (algoritmo, decidibilidade, complexidade, etc) de maneira formal, utilizando problemas como um "plano de fundo" para introduzir os modelos computacionais. Esperam que através dessa estruturação, os professores passem a demonstrar possíveis relações entre problemas fundamentais a Ciência, com questões reais do dia-a-dia, motivando e elucidando a importância de tais conteúdos.

A última abordagem encontrada referente os modelos computacionais descreve um método para adquirir/aperfeiçoar habilidades na criação de modelos teóricos [17]. O

método pode ser utilizado para o ensino de AF, AP, MT e também pode ser estendido para o ensino de gramáticas entre outros tópicos. A proposta visa desenvolver aprendizagem através da construção de jogos, utilizando uma abordagem construcionista. Os autores relatam que com essa abordagem, os estudantes aprendem fazendo, o que potencialmente contribui com a compreensão. Além de permitir que os estudantes tenham participação ativa e poder de personalização, o que pode contribuir com o aumento da motivação e consequentemente com o seu desempenho na disciplina.

3.2.3 Ensino de *Expressões Regulares*

Ezer e Trakhtenbrot [13] relatam a dificuldade de ensinar a caracterização algébrica de linguagens regulares no curso de Teoria dos Autômatos. Os autores relatam o desenvolvimento de uma série de exemplos intuitivos que podem auxiliar os estudantes na compreensão de tais conteúdos e na superação das dificuldades. Esse material é composto por exemplos que demonstram erros típicos em "provas", seguidos de discussões que busca explicar os erros e identificar suas prováveis causas. Com a abordagem os autores esperam envolver os alunos na construção ativa do conhecimento adequado.

A segunda abordagem metodológica encontrada para ensino de expressões regulares na disciplina de Linguagens Formais e Autômatos foi descrita por Arbe, Ortega e Conde [3]. Tal proposta baseia-se em demonstrar aos estudantes a aplicabilidade do conteúdo em diferentes contextos, ressaltando a sua utilização em aplicações web de três camadas. O método apresentado consiste em uma série de exercícios práticos, de modo que os estudantes desenvolvam fragmentos de código em JavaScript, utilizando notação de expressões regulares. Os autores relatam que "em comparação com abordagens mais clássicas, nosso método melhora significativamente a familiarização de alunos com a utilidade prática imediata dos conceitos envolvidos na definição de Linguagens Formais" [3].

3.2.4 Ensino de *Não-Determinismo*

Dois trabalhos foram encontrados referentes a abordagens de ensino para Não-Derterminismo. O primeiro relata a construção de um tutorial com objetivo de familiarizar os leitores do uso do modelo Spin no ensino de conceitos como: concorrência, verificação e não-determinismo [5] . O autor apresenta um ambiente de desenvolvimento integrado (IDE), desenvolvido por ele, chamado o JSpin. E em seguida, apresenta um tutorial que utiliza o JSpin e o *software* de visualização SpinSpider para realizar o desenvolvimento e verificação de programas concorrentes. Uma vez que os programas concorrentes são definidos em termos de Autômatos Finitos Não-Determinísticos, o autor utiliza o desenvolvimento desses programa para ensinar Não-Determinismo.

A segunda abordagem identificada refere-se ao estudo para modificação no ensino do Não-Determinismo [4]. Os autores relatam que em geral a forma de explicação de Autômatos Finitos Não-Determinísticos (AF-ND) pode não ser a mais apropriada. Acredita-se que sem uma abordagem apropriada os alunos tendem a desenvolver uma

imagem de um AF-ND como uma máquina consistente, que percorre todos os caminhos possíveis da árvore até encontrar um caminho de aceitação. Apresenta-se, então, uma intervenção explícita, que pode afetar significativamente os modelos mentais dos alunos de TC E LFA, no sentido de melhorar a percepção do comportamento Não-Determinístico.

3.2.5 Ensino do *Lema do Bombeamento*

Com base na análise da típica dificuldade no ensino do Lema do Bombeamento, Ezer e Trakhtenbrot [12] relata o desenvolvido de uma série de exemplos que possibilitam auxiliar o processo de compreensão desse conteúdo. Esses exemplos são utilizados no curso através de um ambiente virtual de aprendizagem, onde inicia-se discussões conduzidas pelo tutor sobre o raciocínio comum referentes a problemas relacionados com o tema. O objetivo dessas discussões é destacar erros típicos, e encontrar a sua causa. Essa abordagem visa envolver os alunos na construção ativa de compreensão adequada, confrontando seus equívocos.

3.3 Análise das Metodologias Identificadas

Como visto na introdução deste artigo, diversos pesquisadores têm relatado dificuldades no ensino de TC e LFA. Entretanto, com o desenvolvimento dessa pesquisa, identificou-se poucas produções que evidenciem abordagens metodológicas que busquem amenizar tais problemas. Esses dados demonstram que essa é uma área pouco explorada, diante dos problemas que são frequentemente relatados.

Dentre as 15 abordagens encontradas, no geral, os autores têm evidenciado que a forma tradicional de ensino não tem sido suficientemente eficaz para motivar e auxiliar os alunos a desenvolverem um conhecimento adequado e os trabalhos propõem modificações que tornem o estudante mais participativo e os conteúdos mais práticos e relacionados aos temas do dia-a-dia.

Segundo Matos e Silva [19] "a maioria dos artigos apresentados em conferências renomadas da área, como a conferência internacional do SIGCSE – *ACM Special Interest Group on Computer Science Education*, são reflexões acerca de experiências e introspecções dos seus autores". Com o desenvolvimento dessa pesquisa torna-se evidente tal afirmação e ressalta a importância de conceber o Ensino de Computação (ou a Educação em Computação) enquanto área de pesquisa, fazendo com que essa nova área possua um diálogo constante e crítico com as ciências da educação [18].

4 Considerações Finais

Este artigo apresentou 15 abordagens metodológicas descritas na literatura científica entre 2000 e 2014, para o ensino de tópicos referentes à Teoria da Computação, Linguagens Formais e Autômatos em cursos de graduação em Computação. Tais abordagens foram desenvolvidas a fim de amenizar os problemas relacionados ao ensino/aprendizagem da área, buscando melhorar a apreciação dos estudantes quanto ao papel da Teoria da Computação e desenvolver uma aprendizagem ativa.

As metodologias/métodos propostas evidenciam a necessidade de modificar o ensino/aprendizagem dos tópicos relacionados à TC e LFA, tornando os estudantes mais participativos e os conteúdos mais práticos e relacionados aos temas do dia-a-dia. Tais informações ressaltam a importância de conceber o Ensino de Computação (ou a Educação em Computação) enquanto área de pesquisa, fazendo com que essa nova área possua um diálogo constante e crítico com as ciências da educação.

Como trabalho futuro, pretende-se aprofundar as análises por meio de experimentos controlados e avaliação da interação tecnológica com uma ou mais propostas metodológicas. Nesse processo espera-se ao final desta pesquisa apresentar outra(s) possibilidade(s) metodológica(s) para o ensino de conteúdos curriculares de Teoria da Computação, Linguagens Formais e Autômatos.

Referências

1. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195--197 (1981).
2. Aguiar, S. O., Oeiras, J. Y. Y. : Ambiente Moodle de auxílio ao ensino e aprendizagem em Linguagens Formais. XVIII Workshop sobre Educação em Computação, 818--827, (2010).
3. Arbe, J. M. B., Ortega, A. S., Conde, J. I. M.: Formal languages through web forms and regular expressions. ACM SIGCSE Bulletin, v. 39, n. 4, 100--104 (2007).
4. Armoni, M., Lewestein, N., Ben-Ari, M.: Teaching students to think nondeterministically. ACM SIGCSE Bulletin, 4--8 (2008).
5. Ben-Ari, M.: Teaching Concurrency and Nondeterminism with Spin. Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education, 363--364 (2007).
6. Borges, M.: Avaliação de uma metodologia alternativa para a aprendizagem de programação. VIII Workshop de Educação em Computação-WEI (2000).
7. Brasil.: Diretrizes Curriculares Nacionais para os cursos de graduação em Computação. http://portal.mec.gov.br/index.php?option=com_content&id=12991:diretrizes-curriculares-cursos-de-graduacao.
8. Chakraborty, P., Saxena, P. C., Katti, C. P.: Fifty years of automata simulation: a review. ACM Inroads, v. 2, n. 4, 59--70 (2011).
9. Chakraborty, P., Saxena, P. C., Katti, C. P.: Automata simulators: Classic tools for computer science education. British Journal of Educational Technology, v. 43, n. 1, 2011—2013 (2012).
10. Chesñevar, C. I., González, M. P., Maguitman, A. G.: Didactic strategies for promoting significant learning in formal languages and automata theory. ACM SIGCSE Bulletin, v. 36, n. 3, 7--11 (2004).
11. Dognini, M. J., Luís, A., Raabe, A.: EduLing - Software Educacional para Linguagens Regulares. XIV Simpósio Brasileiro de Informática na Educação – NCE, IM/UFRJ (2003).
12. Ezer, J. G., Trakhtenbrot, M.: Challenges in teaching the pumping lemma in automata theory course. ACM SIGCSE Bulletin, v. 37, n. 3, 369 (2005).
13. Ezer, J. G., Trakhtenbrot, M.: Algebraic Characterization of Regular Languages: How to Cope With All These Equivalences ? ACM SIGCSE Bulletin, v. 38, n. 3 (2006).
14. Furtado, O. J. V.: O ensino de Linguagens Formais vinculado ao ensino de Compiladores. XI Workshop de Educação em Computação (2003).

15. Hopcroft J. E., Motwani, R., Ullman, J. D.: Introduction to Automata Theory, Languages and Computation. 3rd Editio (2006).
16. Knobelsdorf, M., Kreitz, C., Bohne, S.: Teaching Theoretical Computer Science using a Cognitive Apprenticeship Approach Categories and Subject Descriptors. SIGCSE '14: Proceedings of the 45th ACM technical symposium on Computer science education, 67--72 (2014).
17. Korte, L., Anderson, S., Good, J., Pain, H.: Learning by Game-Building : A Novel Approach to Theoretical Computer Science Education. 12th annual SIGCSE conference on Innovation and technology in computer science education, v. 39, 53--57 (2007).
18. Lister, R.: Teaching-oriented faculty and computing education research. ACM Inroads, v. 3, n. 1, 22--23 (2012).
19. Matos, E., SILVA, G. da.: Currículo de licenciatura em computação: uma reflexão sobre perfil de formação à luz dos referenciais curriculares da SBC. Anais do XXXII Congresso da Sociedade Brasileira de Computação - XX Workshop sobre Educação em Computação - WEI (2012).
20. Merceron, A.: Design Patterns to Support Teaching of Automata Theory. ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education, 60558 (2009).
21. Pirovani, J. C.; Mataveli, G. V.: Estudo e adaptação de software para o ensino de Linguagens Formais e Autômatos. Revista Brasileira de Informática na Educação, v. 21, 53--68, (2014).
22. Ruehr, F.: Strategies in the theory of computation. Journal of computing sciences in colleges, v. 17, n. 2, 93--105 (2001).
23. Sá, C. C. DE, Bittencourt, G.: Uma Proposta para Disciplina de Teoria da Computação. XII Workshop sobre Educação em Informática (2004).
24. Scarton, C. E., Aluisio, S.: O uso do MERLOT por Alunos de Teoria da Computação para a Criação de Materiais de Ensino-Aprendizagem. XIX Workshop sobre Educação em Computação (2011).
25. Sigman, S.: Engaging Students in Formal Language Theory and Theory of Computation. SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education, 450--453 (2007).
26. Sipser, M.: Introduction to the Theory of Computation. 2nd Editio (2005).
27. Verma, R. M.: A visual and interactive automata theory course emphasizing breadth of automata. ACM SIGCSE Bulletin, v. 37, n. 3, 325--329 (2005).
28. Vijayalaskhmi, M.; Karibasappa, K.: Activity based teaching learning in formal languages and automata theory-An experience (2012).
29. Parker, G.: Getting more from Moore. Primus, v. 2, 235--246 (1992).

Calculation and Applications of Concurrent Rules

Jonas Santos Bezerra, Leila Ribeiro

Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
Email: {jsbezerra,leila}@inf.ufrgs.br

Abstract. Graph Grammars are a suitable formalism to model and reason about complex systems. In this paper, we will review a particular static analysis technique, the calculation of concurrent rules, that can be used to capture the cumulative effects of applying several rewriting rules, which helps determine whether a graph grammar behaves as intended. Also, we show the problem of combinatorial explosion that can occur when calculating the concurrent rules and some strategies that can be used to manage it by constraining the problem domain and filtering undesired rules.

1 Introduction

Graph Grammars are a suitable formalism to model and reason about complex systems, providing both a visual modelling language and formal analysis techniques [7]. The states of a system are modelled as graphs, while the actions which can alter these states are modelled as graph transformation rules that work by applying local modifications into graphs.

Given its formalism, there are several analysis techniques that can be performed over a Graph Grammar, including the *concurrent rule* construction, which can be used to summarize the application of a sequence of several rules in one single step.

In this paper, we specifically show how to construct concurrent rules to help the modeller of a system to check whether the designed model behaviours as intended, and also which kind of unexpected behaviours could emerge from the system, specially when dealing with concurrency.

Also, we address the problem of combinatorial explosion inherent to the construction of concurrent rules by enriching the graph grammar with negative application conditions and graph constraints, which are used to restrict the problem domain. The concurrent rule construction and the strategies presented here are being implemented (together with other analysis techniques) in Verigraph[1]: A software specification and verification tool based on graph rewriting.

[1] Source Code: `github.com/verites/Verigraph`,
Documentation: `verites.github.io/Verigraph`

This work is organized as follows: In section 2 we review the basic definitions under the graph transformation theory. Section 3 introduces a running example that will be used to demonstrate the concepts and constructions presented in this paper. Section 4 presents the *concurrent rule* analysis technique, after that section 5 presents the strategies to address the problem of combinatorial explosion while filtering more meaningful concurrent rules. Finally, section 6 presents the conclusions and future work.

2 Algebraic Graph Transformation

In this section, we review the basic definitions of algebraic graph transformation according to the double pushout approach [3]. These definitions are standard in the area, and more details can be found in [2].

A *graph* is a tuple $G = (V, E, s, t)$ where: V is a set of nodes, E is a set of Edges and $s, t : E \to V$ are two total functions that map each edge in E to its source and target in V.

Given two graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for i in $[1, 2]$, a *graph morphism* $f : G_1 \to G_2$ between them is a pair $f = (f_V, f_E)$ where $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ are total functions that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$.

A *type graph* is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ where V_{TG} and E_{TG} are called the node and edge type alphabets, respectively.

A *typed graph* is a pair $G^T = (G, type)$ consisting of a graph G and a graph morphism $type : G \to TG$.

Given two typed graphs G_1^T, G_2^T with $G_i^T = (G_i, type_i)$ for i in $[1, 2]$, a *typed graph morphism* $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \circ f = type_1$.

Unless explicitly stated otherwise, for the rest of this paper we will work with typed graph and typed graph morphisms only, thus we will omit the mention to the typing where it does not damage the content.

In the DPO approach, a graph rule[2] $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is a span of injective graph morphisms $l : K \to L$ and $r : K \to R$ where the graphs L, K and R are called the left-hand side (lhs), gluing graph and right-hand side (rhs).

Graph rules are used to locally modify graphs in a process called *graph transformation*: when a morphism from the lhs of a rule is found on a graph, we delete the elements of L that are not in K and add the elements of R that are not in K, as long as the result is still a graph (elements in K are always preserved).

Given a graph transformation rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a graph G with a graph morphism $m : L \to G$, called match, a *direct graph transformation* $G \overset{p,m}{\Longrightarrow} H$ from G to a graph H is a double-pushout (DPO) diagram in the category $Graphs_{TG}$.

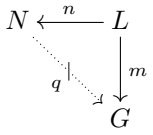
$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle k} & (2) & \downarrow{\scriptstyle m'} \\ G & \xleftarrow{\ f\ } & D & \xrightarrow{\ g\ } & H \end{array}$$

[2] Also called graph transformation rule or graph production.

The existence of the DPO-diagram, and consequently of the graph transformation, depends on the satisfiability of the so called *gluing conditions*: (*i*) *identification condition*, the match does not identify a deleted element with another that is preserved or also deleted; (*ii*) *dangling condition*, the match does not delete a node without deleting all of its incident edges.

Besides the match and the gluing conditions, other constructs can be used to impose more conditions to the existence of a graph transformation, here we present two of such constructs: *negative application conditions* and *graph constraints*.

A *(left) negative application condition* (NAC for short) over a graph rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is of the form $NAC(n)$, where $n : L \to N$ is an arbitrary graph morphism. A match $m : L \to G$ of a rule p satisfies $NAC(n)$ on L (written $m \models NAC(n)$) iff $\nexists\, q : N \to G$ with q injective and $q \circ n = m$.



Given a set $NAC_L = \{NAC(n_i) | i \in I\}$ of left $NACs$, a match $m : L \to G$ satisfies NAC_L, iff $m \models NAC(n_i)\ \forall i \in I$. The *right NACs* are defined analogously for the right-hand side of a graph rule and the comatch $m' : R \to H$ of the correspondent graph transformation.[3]. A *graph rule with NACs* is then a pair (p, N) where p is a production and N is a set of NACs for p.

A *positive atomic constraint* is of the form $PC(a)$, where $a : P \to C$ is a graph morphism. A graph G satisfies $PC(a)$ if for every injective graph morphism $p : P \to G$ there is at least one injective graph morphism $q : C \to G$ such that $p = q \circ a$. A *negative atomic constraint* is defined analogously, but the satisfiability condition is that there is no injective graph morphism $q : C \to G$ such that $p = q \circ a$. A *graph constraint* is then defined as a boolean formula over atomic constraints.

A *typed graph grammar with NACs and graph constraints* is a tuple $GG = (TG, G_0, P, C)$ where TG is the type graph, G_0 is the initial graph (typed over TG), P is a set of productions with NACs and C is a set of graph constraints. We will call it *graph grammar* for short.

3 Modelling with Graph Grammars

We use a Graph Grammar that models a client-server scenario for an e-mail application to illustrate our examples. This system has four actions: (*i*) $sendMessage$: client sends message to server, (*ii*) $getData$: data is obtained from server, (*iii*) $receiveMessage$: server sends message to client, (*iv*) $deleteMessage$: client obtains data from received message / message is destroyed.

The rules for this grammar are depicted in Fig 1, the *gluing graphs* are omitted, but can be seen as the nodes with the same figures and edges with the same numbers that appear in both LHS and RHS of each rule (a number before a colon represents the mapping of an edge that appears in both sides, while a

[3] Verigraph takes only rules with left NACs as input, however they can always be transformed into right NACs, which is necessary for our concurrent rule construction.

number after a colon or without a colon represents the type of the edge). The type and initial graphs were omitted due to lack of space[4].

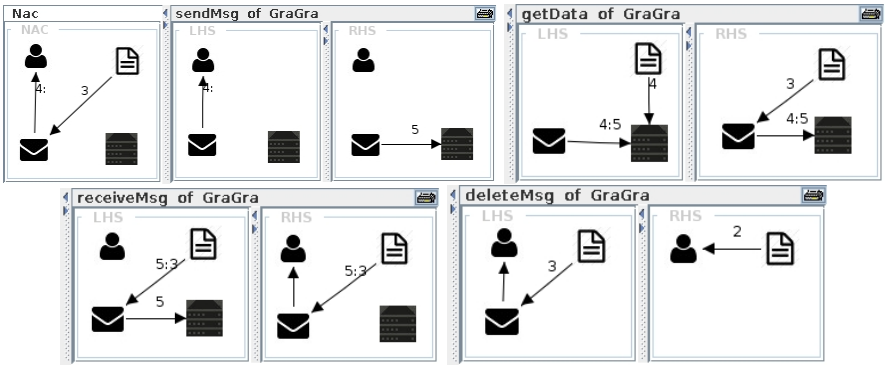


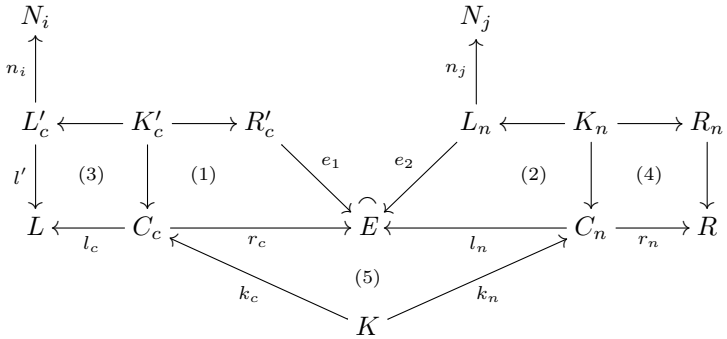
Fig. 1. Graph rules for Server

4 Concurrent Rules

Graph Grammars that represent real systems usually have a considerable number of rules, possibly making it difficult to the modeller to foresee all possible rule interactions. Therefore it is important to have analysis techniques to address this issue, as well as tools that implement them.

We said earlier that the aim of calculating the concurrent rules is to summarize the combined effects of applying the transformations induced by a sequence of graph rules. Here we present how this can be done.

A *rule sequence* is a list containing rules of a grammar in an specific order in which the modeller wants them to be applied. Given a rule sequence $r = p_0, \ldots, p_{n-1}, p_n$, the construction of its correspondents concurrent rules is done by recursively combining pairs of subsequent rules, where the pairwise combination is defined as follows [2,4]:



For $n = 0$**:** The *concurrent rule* p_c for the single rule p_0 is p_0 itself. If p_0 has a set of NACs it will be preserved.

[4] The complete grammar definition can be found together with Verigraph source code.

For $n \geqslant 1$**:** A concurrent rule $p_c = p'_c *_E p_n$ with NACs for the rule sequence $p_0, \ldots, p_{n-1}, p_n$ is defined as $p_c = (l_c \circ k_c : K \to L, r_n \circ k_n : K \to R)$ where:
- $p'_c : L'_c \leftarrow K'_c \to R'_c$ is a concurrent rule for the sequence $p_0, \ldots, p_{n-1}$
- E is an overlapping of R'_c and L_n with (e'_c, e_n) jointly surjective
- (1)-(3) and (2)-(4) are valid double-pushout rewritings
- (5) is a pullback
- All the NACs N_i of the production p'_c are shifted over the morphism l', resulting in a set of NACs $n'_i : L \to N'_i$
- All the NACs N_j of the production p_n are shifted over the morphism e_2 and then over the span $q'_c = l_c : C_c \to L, r_c : C_c \to E$, resulting in a set of NACs $n'_j : L \to N'_j$ [5]

4.1 Example: Concurrent Rules for $sendMessage + getData$

Given the rule sequence: $sendMessage + getData$ from the rules depicted in Fig 1, we want to generate the concurrent rules that summarize the behaviour of this sequence in one step.

First, we generate all possible jointly surjective pairs (overlappings) between the rhs of $sendMessage$ and the lhs of $getData$. All possible combinations are shown in Fig 2.

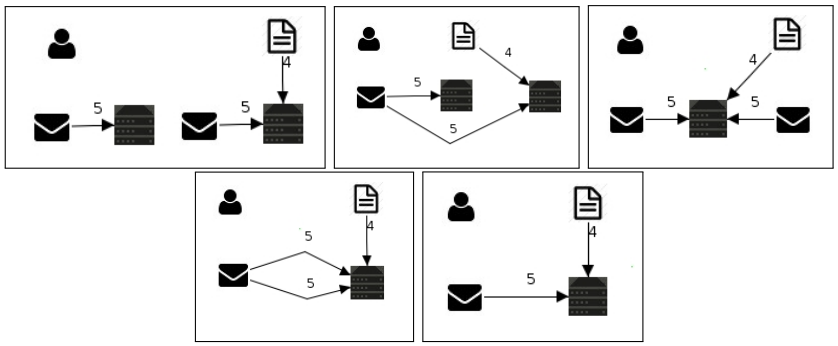


Fig. 2. Overlappings for $sendMessage$ and $getData$.

After that, we check which overlappings satisfy the gluing conditions of $sendMessage^{-1}$ and $getData$. In this particular case they all do, so each one of them will generate a different concurrent rule.

For each overlapping pair that satisfies those gluing conditions, we proceed to the calculation of the pushouts and pullbacks that will result in the concurrent rule. Fig 3 presents the diagram for calculating the concurrent rule correspondent to the last overlapping[6] depicted in Fig 2. Note that it is a concrete instance of the diagram presented before.

The concurrent rule in our example is defined by the bottom of the diagram in Fig 3, and it represents the act of sending a message and getting a piece of

[5] Both the algorithms for shifting NACs over a morphism and over a span are omitted here due to lack of space, but can be found in [4].

[6] Due to space limitations we do not show the concurrent rules for every pair.

data when there is only one element of each type present in the system state. The other overlappings show other possible situations, some of them concurrent, that could occur when the rules are not necessarily applied to the same elements.

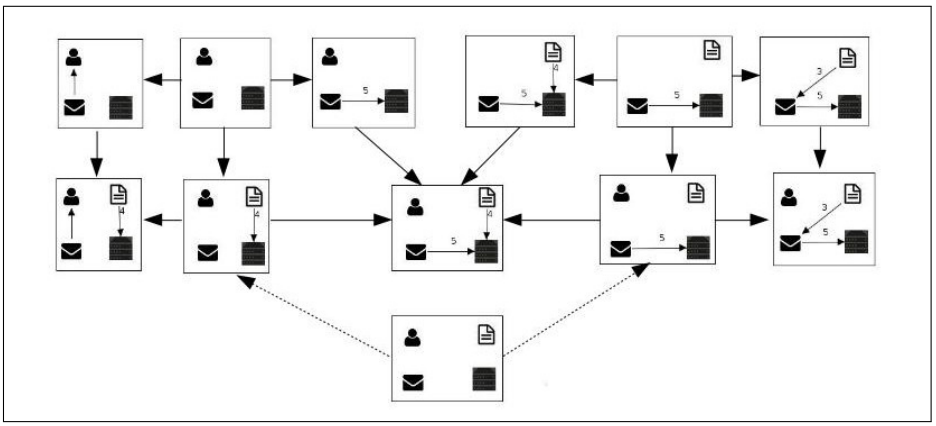


Fig. 3. One possible concurrent rule for $sendMessage$ and $getData$.

Verigraph allows for the calculation of all possible concurrent rules. Nonetheless, this is an expensive operation, since there exists a concurrent rule for each different overlapping between a R'_c and a L_n in each step of the induction, which may result in a combinatorial explosion. For example, if we were to compute all the concurrent rules for the sequence $sendMessage+getData+receiveMessage$, we would have to calculate all possible overlappings for the rhs of each rule already calculated for $sendMessage + getData$ with the lhs of $receiveMessage$.

Also, the shifting of NACs adds considerably to this cost. In next section, we present the strategies used which are being used in verigraph in order to reduce this costs.

5 Dealing with the combinatorial explosion

There exist some strategies that can help dealing with the combinatorial explosion of by addressing some specificities of the problem domain. The strategies explained in the following do not solve the theoretical worst cases, but they have been showed to be good enough in most of our practical cases.

5.1 Trivially-triggered NACs

For each pair of rules (p_c, p_n) for which we want to generate the corresponding concurrent rules, we must first generate all possible overlappings between R_c and L_n, check whether they satisfy the gluing conditions and calculate the pushouts and pullbacks that will result in the concurrent rules. However, it is possible

that some of the generated overlappings result in epimorphic pairs $(E, e_1 : R_c \to E, e_2 : L_n \to E)$ whose morphisms e_1 or e_2 do not satisfy the right NACs of p_c or the left NACs of p_n, respectively.

In such cases, the NACs forbid the existence of valid transformations $L \overset{p_c,l'}{\Longrightarrow} E$ and $E \overset{p_n,r'}{\Longrightarrow} R$ even though the gluing conditions are satisfied. It means that the rules could not be applied over the graph E. We may them ignore such overlappings when calculating possible concurrent rules.

If we do maintain those pairs for computation, as the shift of NACs aims to translate the NACs of each rule to sets of equivalent NACs in the concurrent rules, we would generate rules where for every possible match of L, there will always be a NAC not satisfied by m, thus the rule would never be applicable.

5.2 Graph Constraints

Graph constraints can be used to globally enforce or prohibit the existence of certain structures in the graphs that can be generated by a graph grammar. For example, they can be used to define minimal and maximal multiplicities for nodes and edges.

When calculating the concurrent rules for a pair (p_c, p_n) we first use them similarly to the use of NACs, checking whether the generated overlappings satisfy the graph constraints, cutting off those who do not, which can also lead to a reduction of possible concurrent rules. Fig 4 shows two negative atomic constraints that (a) forbid the existence of more than one server and (b) forbid a piece of data of being in two different messages at the same time. Look at Fig 2 again to see that two of the overlappings would be cut off by these constraints.

We can still use the graph constraints to cut off even more concurrent rule candidates, because even though the overlappings satisfy the gluing conditions and NACs, the resulting lhs and rhs may still not satisfy the graph constraints.

When dealing with injective morphisms, if a graph L of a rule does not satisfy the graph constraints, no possible $m : L \to G$ can be found in which G satisfies the constraints, thus the rule can never be applied and we can discard it. Similar reasoning can be applied to the R graph and $m' : R \to H$.

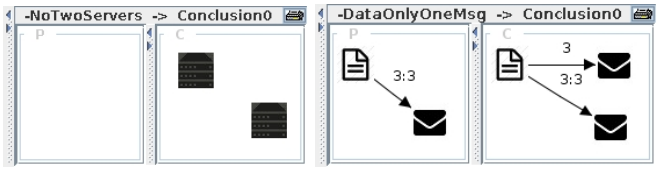


Fig. 4. Negative atomic constraints for server

5.3 Concurrent Rules Induced by Dependencies

The default algorithm constructs the concurrent rules based on all the overlappings of the right side of the first rule and left side of the second rule, allowing

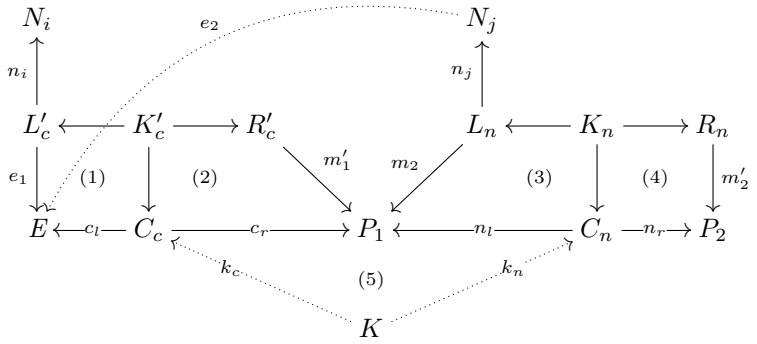
us to see how the elements created/preserved by one rule can be connected with the elements deleted/preserved by the other.

One way to restrict the number of possible concurrent rules and still generate meaningful rules is to filter and use only the overlappings associated with dependencies [5] between subsequent pairs of rules. The idea behind this is to use only the overlappings where (1) the elements needed for the second rule to be applied are explicitly created by the first one or (2) the elements forbidden by the NACs of the second rule are explicitly deleted by the first.

For the first case, we only need to filter the overlappings in the concurrent rule diagram where $\nexists h_{12} : R'_c \to C_n$ such that ($l_n \circ h_{12} = e_1$ and $r_n \circ h_{12} \models N_i^{-1}$) or $\nexists h_{21} : L_n \to C_c$ such that ($r_c \circ h_{21} = e_2$ and $l_c \circ h_{21} \models N_j$).

However, this notion does not take into consideration the elements whose existence is forbidden by the NACs of the second rule and would then forbid its application, but once deleted by the first rule the application of the second is enabled.

We did not find on the literature a construction to capture those cases, however an adaptation of the concurrent rule algorithm can be made based on the algorithm for calculating dependencies between the rules defined in [5]. Besides the overlappings between (R'_c, L_n) that represent dependencies, we generate *also* the overlappings of the left side of the first rule L'_c with the NACs N_j and check whether the rewritings are possible, in which case the diagram for the corresponding concurrent rule construction can be seen as follows:



5.4 Maximal Concurrent Rule

Sometimes, instead of generating all possible overlappings or even all the dependencies, the modeller is interested in seeing only the maximal interactions between the elements of each rule in the sequence, thus we may filter the overlappings with the least number of elements, capturing the cases where the elements of each rule are as connected as possible. Note that the rule in Fig 3 is a maximal concurrent rule.

5.5 Comparison

Here we present some of the results when calculating the concurrent rules with and without the strategies presented. The grammars used are shipped together with Verigraph source code.

On table 1 we show the results for the grammar in section 3, where we calculated the concurrent rules for it complete execution, represented by the rule sequence: $sendMsg + getData + receiveMsg + deleteMsg$. The constraints used were the ones presented in section 5.2.

Strategy	**Time**	**Number of Rules**	**Number of NACs**
None	1m30s	1021 rules	$\approx$ 4 nacs/rule
Constraints on Overlappings	8s	141 rules	$\approx$ 2 nacs/rule
Constraints on LHS and RHS	8s	141 rules	$\approx$ 2 nacs/rule
Dependencies	0.45s	2 rules	2 nacs/rule
Maximal Rule	0.32s	1 rule	2 nacs/rule

Table 1. Concurrent rules for the server grammar

On table 2, we show the results for a grammar representing the behaviour of an elevator system, where we calculated the concurrent rules that generate states where there are four floors and the elevator has to attend a call request in one of these floors. The corresponding rule sequence is: $AddFloor + AddFloor + InitialHigher + InitialHigher + InitialHigher + AddTransitiveHigher + AddTransitiveHigher + AddTransitiveHigher + AddRequest$. Fifteen graph constraints where used to restrict invalid states such as: the elevator can not be on two floors at the same time; two floors can not hold the same request; if floor a is higher than b, then b is not higher than a, among other possibilities.

When calculating the concurrent rules for the elevator grammar without any strategy, verigraph used all the memory available in the computer but even after more than two days running it did not provide an answer, so we halted the program. For the case of Maximal Rule, verigraph provided zero concurrent rules because in some step of the computation the least disjoint overlapping(s) did not satisfy some constraints, thus being discarded.

Strategy	**Time**	**Number of Rules**	**Number of NACs**
None	>2d	NA	NA
Constraints on Overlappings	30m	11313 rules	$\approx$ 0.7 nacs/rule
Constraints on LHS and RHS	1m55s	287 rules	$\approx$ 5 nacs/rule
Dependencies	2m15s	9 rules	$\approx$ 4 nacs/rule
Maximal Rule	0.93s	0 rules	0 nacs/rule

Table 2. Concurrent rules for elevator grammar

6 Conclusions

We presented a statical analysis technique for Graph Grammars called concurrent rules, used to summarize the behaviours of several rules into a single step in order to understand the overall behaviour a system.

We also presented some strategies that help to deal with the combinatorial explosion inherent to this problem. The technique and strategies shown are being implemented along with others in the Verigraph tool, under development at the Instituto de Informática, Universidade Federal do Rio Grande do Sul.

Verigraph is already being used to support a methodology for systematically checking Use Cases and other action-driven textual documents in order to identify and remove problems such as inconsistencies, omissions and ambiguities,

as described in [6,1]. In this setting, the analysis of concurrent rules is used to check whether the global aims of the documents are fulfilled as well as identify unexpected side effects.

It is important to recall that these strategies do not solve the combinatorial explosion problem at all, they only allow us to cut off rules that are not interesting to the modeller or that violate an assumption of the grammar as soon as possible, before they start to propagate.

This is part of a larger ongoing project, where we are investigating not only how to verify the behaviour of textual documents and systems modelled as Graph Grammars, but also how to use these models to generate software tests to cover the system, avoiding test redundancy and, as much as possible, the combinatorial explosion.

Acknowledgements

The authors would like to acknowledge the brazilian agencies CNPq, CAPES and FAPERGS for their support in the form of financial aid (VeriTes project/CNPq) and scholarships (CNPq).

References

1. Jonas Santos Bezerra, Andrei Costa, and Leila Ribeiro. Formal Verification of Health Assessment Tools : a Case Study. *Electronic Notes in Theoretical Computer Science*, 324(WEIT 2015, the Third Workshop-School on Theoretical Computer Science):31–50, 2016.
2. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
3. H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 167–180, Oct 1973.
4. Leen Lambers. *Certifying rule-based models using graph transformation.* PhD thesis, Berlin Institute of Technology, 2009.
5. Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations: Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, pages 61–76, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
6. Marcos Oliveira, Leila Ribeiro, Érika Cota, Lucio Mauro Duarte, Ingrid Nunes, and Filipe Reis. *Use Case Analysis Based on Formal Methods: An Empirical Study*, pages 110–130. Springer International Publishing, Cham, 2015.
7. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

The Smix synchronous multimedia language: Operational semantics and coroutine implementation

Guilherme F. Lima[1], Christiano Braga[2], and Edward Hermann Haeusler[1]

[1] PUC-Rio, Rio de Janeiro, Brazil
{glima,hermann}@inf.puc-rio.br
[2] UFF, Niterói, Brazil
cbraga@ic.uff.br

Abstract. Smix is a domain-specific language for the construction of interactive multimedia presentations. Its programs describe how media objects (texts, images, videos, etc.) should be presented and how external events, such as the passage of time or user interaction, affect their presentation. What distinguishes Smix from similar high-level multimedia languages, such as NCL, SMIL, and HTML, is first is simplicity: the language has only three main concepts (media object, action, and link) which can nonetheless be used to program complex multimedia applications. The second distinguishing characteristic of Smix is its synchronous, deterministic semantics, which induces a precise notion of logical time. In this paper, we introduce the Smix language, present two versions of its synchronous semantics, equational and linear, both in big-step operational style, and discuss a novel, straightforward implementation of its linear semantics using Lua coroutines.

1 Introduction

Smix [10] (Synchronous Mixer) is a domain-specific language for the construction of interactive multimedia presentations. Its programs use synchrony relations (links) to describe how media objects (texts, images, videos, etc.) should be presented and how external events, such as the passage of time or user interaction, affect their presentation.

There are two characteristics that distinguishes Smix from similar high level multimedia languages, such as NCL, SMIL, and HTML.[3] The first one is simplicity. Smix has only three main concepts, namely, media, action, and link, which can nonetheless be used to program complex multimedia applications. A simpler language implies in a simpler semantics and, consequently, a simpler implementation. NCL, SMIL, and HTML, in contrast, are huge languages with numerous concepts and constructions to represent them. Despite its simplicity, most NCL concepts can be easily simulated in Smix, as discussed in [10]; in fact, the Smix language was deliberately designed to serve as an abstraction layer (the language of a multimedia virtual machine) for the implementation of other higher-level multimedia languages, in particular, Plain Smix (a syntactically richer dialect of Smix), NCL, and to a lesser extend, SMIL.

[3] NCL is the standard declarative language for interactive applications in the Brazilian digital terrestrial television system [1] and an ITU-T recommendation for IPTV applications [8]. SMIL is a widely adopted W3C recommendation [16] for interactive multimedia presentations. And HTML is a W3C recommendation [17] (and core Web technology) for typesetting hyperlinked text together with images, and more recently, audio and video.

The second distinguishing characteristic of Smix is its deterministic, synchronous semantics, which gives its programs a precise notion of logical time. The semantics of NCL, SMIL and HTML, in contrast, is notoriously complex and obscure, especially in relation to time [10]. Even on a logical level, these languages treat time as something external to the system. Its representation and manipulation can be influenced by physical phenomena, such as processing or communication delays, which are unpredictable or implementation dependent, and which can thus lead to nondeterminism and dyssynchrony. Smix, on the other hand, is a synchronous language with a semantics that guarantees determinism and logical correctness. By calling it synchronous, we mean that its programs operate under the *synchronous hypothesis* [5], i.e., that they can be viewed as input-driven systems whose reactions are instantaneous. The synchronous hypothesis induces a precise notion of logical time in which the only relevant concepts are those of simultaneity and precedence between events.

In [15] the authors propose a rewriting-logic semantics for NCL, and in [14] the author proposes an authoring language-independent model for multimedia documents. There are other proposals of formal semantics for NCL [12] and similar proposals for SMIL [4]. Most of these works, however, are concerned not with the implementation of interpreters (which is the main goal of Smix) but with static validation of program properties, usually within a larger system of user-guided verification. Their models tend to be complex and impractical, especially if real-time performance is needed.

In this paper, we focus on primarily the formalization of the synchronous semantics of Smix, and present two versions of it: the equational semantics and the linear semantics. Both versions follow the operational approach to semantics [13]; the particular style used is that of big-step (or natural) operational semantics [9]. Both formalizations were inspired by the formal semantics of the synchronous language Esterel [3,2], and as such are only concerned with the description of a single program reaction.

The rest of the paper is organized as follows. In Section 2, we introduce the Smix language and discuss the intuitive behavior of its programs. In Section 3, we present the equational semantics, which formalizes this intuitive behavior. The problem with the equational semantics is that it does not guarantee termination in bounded time, which violates the synchronous hypothesis. In Section 4, we present the linear semantics that solves this problem by adopting a linear format for programs, which replaces the equational format, and in which reactions always execute in bounded time. In practice, before executing a program, the Smix interpreter "linearizes" it, i.e., converts it from the equational to the linear format. In Section 5, we present a simple implementation of the linear semantics in a Lua [7] multimedia library augmented with coroutines (Section 5). Finally, in Section 6 we draw our conclusions and point out future work.

2 The Smix language

Smix is a high-level declarative language for the construction of multimedia presentations. Its goal is to offer simple but expressive abstractions for the precise representation of complex audiovisual ideas. A Smix program is a set of media object declarations together with a sequence of links. A media object is a presentation atom (e.g., image, audio, video, etc.) and has associated with it an identifier, a content, a state, a time, and a property table.

The identifier uniquely identifies object in the program. The content is a possibly empty sequence of audiovisual samples. The state is either "occurring" (playing), "paused", or "stopped". The time is the number of clock ticks to which the object was exposed while in state occurring. And the property table maintains the object properties—their value determine the characteristics of the object's presentation, e.g., the value of property "transparency" determines the transparency applied to its visual samples.

In Smix, media objects are manipulated by actions. There are five possible actions: start ($\triangleright$), stop ($\square$), pause ($\blacksquare\!\blacksquare$), seek ($\twoheadrightarrow$), and attribution ($\circ$). The first three actions, start, stop, and pause, manipulate the object's state; the last two, seek and set, manipulate the object's time and property table. Actions have the general form (predicate ? target : argument), where the predicate is a propositional logic formula involving the state, time, or property values of media objects, the target specifies the operation ($\triangleright$, $\square$, $\blacksquare\!\blacksquare$, $\twoheadrightarrow$, or $\circ$) and main operand (media object or property) of the action, and the argument is an extra operand (expression) required by seek and set actions.

The execution of an action is conditioned by the validity of its predicate. To evaluate an action, the interpreter (more precisely, the language kernel) first evaluates its predicate. If it is false, the action is discarded; otherwise, if it is true, the kernel proceeds to execute the action: it evaluates the extra argument (if any) and tries to execute the specified operation with the given operands. When writing actions, we often omit the predicate, question mark, and parentheses when the predicate is tautological (always true). Thus (i) an action of the form $\triangleright x$, read "start x", when executed, puts x in state occurring; (ii) an action of the form $\blacksquare\!\blacksquare x$, read "pause x", puts x in state paused; (iii) an action of the form $\square x$, read "stop x", puts x in state stopped; (iv) an action of the form $\twoheadrightarrow x : e$, read "seek x by e", advances the playback time of x by the number to which expression e evaluates; and (v) an action of the form $\circ x.u : e$, read "set $x.u$ to e", stores into property u of x the value to which expression e evaluates.

A Smix program consists of two parts: a set of media object declarations and a sequence of links. A media object declaration associates an object identifier with a property initialization table. A link is a synchrony relation of the form $a \rightarrow a_1 a_2 \ldots a_n$, which establishes that whenever some action with target a is executed, actions $a_1, a_2, \ldots, a_n$ shall also be executed, in this order. The action target a on the left-hand side of symbol $\rightarrow$ is called the head of the link, and the action sequence $a_1 a_2 \ldots a_n$ on its right-hand side is called the tail of the link.

Example. To make matters concrete, consider the following Smix program:

$$\triangleright\lambda \rightarrow \triangleright x$$
$$\triangleright x \rightarrow \triangleright y \, \square z$$
$$\triangleright y \rightarrow \triangleright z$$
$$\square x \rightarrow \square\lambda$$

This program has four links which operate on four media objects: the ordinary objects x, y, and z, and the implicit object lambda (λ) which stands for the program itself. The first link establishes that when the program starts, media object x shall be started; the second link establishes that whenever x starts, object y shall be started and object z shall be stopped; the third link establishes that whenever y starts, object z shall be started; and the fourth link establishes that when x stops the whole program shall be stopped.

The equational and linear semantics discussed in Sections 3 and 4 and are only concerned with the description of a single program reaction (input-output cycle). Given some input action a received from the environment, they determine how the execution of action a affects the kernel memory (state, time and properties of media objects) and the actions $a_1, a_2, \ldots, a_n$ that are to be triggered internally in response to a, and emitted back to the environment at the end of the reaction.

3 The equational semantics

Smix has the following syntactic sets: integers $n \in \mathrm{N}$; truth values $t \in \mathrm{T} = \{\top, \bot\}$; media object identifiers $x, y, z \in \mathrm{Media}$; property identifiers $u, v \in \mathrm{Prop}$; expressions $e \in \mathrm{Expr}$; predicates $p \in \mathrm{Pred}$; action atoms $a \in \mathrm{ActAtom}$; action sequences $\alpha \in \mathrm{ActSeq}$; and link sequences (or programs) $L, P \in \mathrm{LinkSeq}$. Its abstract syntax is defined as follows:

$$
\begin{aligned}
e \in \mathrm{Expr} &::= n \mid \mathrm{state}(x) \mid \mathrm{time}(x) \mid \mathrm{prop}(x,u) \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \div e_2 \\
p \in \mathrm{Pred} &::= \top \mid \bot \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \neg p_1 \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \\
a \in \mathrm{ActAtom} &::= (p\,?\triangleright x) \mid (p\,?\,\mathrm{II}\, x) \mid (p\,?\,\square x) \mid (p\,?\twoheadrightarrow x{:}e) \mid (p\,?\,\circ x.u{:}e) \\
\alpha \in \mathrm{ActSeq} &::= \varepsilon \mid a\alpha_1 \\
L \in \mathrm{LinkSeq} &::= \varepsilon \mid \triangleright x \to \alpha L_1 \mid \mathrm{II}\, x \to \alpha L_1 \mid \square x \to \alpha L_1 \mid \twoheadrightarrow x \to \alpha L_1 \mid \circ x.u \to \alpha L_1
\end{aligned}
$$

The program state is represented by a media memory, i.e., a total function θ that maps a media object identifier x to a memory cell $\langle s, n, \rho\rangle$, where $s \in \{\triangleright, \mathrm{II}, \square\}$ is the object state, $n \in \mathrm{N}$ is its time, and $\rho\colon \mathrm{Prop} \to \mathrm{N}$ is a total function that represents its property table. We write $\mathcal{M}$ for the set of all media memories, ϕ for the empty memory cell $\langle \square, 0, \rho_0\rangle$, where ρ_0 is the table in which all properties have value 0, and Φ for the empty memory, i.e., the one in which all cells are empty.

Memory cells can be read and written. Given a memory θ and a media object x, we write $\theta(x)$ for the cell of x in θ and $\theta[x := X]$ for the memory obtained by replacing $\theta(x)$ by X. We write $\theta_s(x)$, $\theta_t(x)$, $\theta_\rho(x,u)$ for the state, time, and value of property u of x in θ, and $\theta_s[x := s]$ for the memory obtained by replacing $\theta_s(x)$ by s, $\theta_t[x \mathrel{+}= n]$ for the memory obtained by incrementing $\theta_t(x)$ by n, and $\theta_\rho[x.u := n]$ for the memory obtained by replacing $\theta_\rho(x,u)$ by n.

Finally, to access the links of a program, we define the link function ℓ that receives as arguments the program P and an action atom a, and returns the action sequence α associated with the execution of a in P ($\tau(a)$ denotes the target of action a):

$$
\begin{aligned}
\ell(\varepsilon, a) &= \varepsilon \\
\ell(a' \to \alpha L, a) &= \begin{cases} \alpha\ell(L,a) & \text{if } \tau(a) = a' \\ \ell(L,a) & \text{otherwise}\,. \end{cases}
\end{aligned}
$$

Evaluation of equational programs. The evaluation of action sequences is determined by the relation $\Rightarrow$ such that $\langle \alpha, P, \theta\rangle \Rightarrow \theta'$ iff action sequence α when executed over program P in memory θ evaluates to the updated memory θ'. Since program P remains fixed throughout the evaluation, we use the notation $\langle \alpha, \theta\rangle \Rightarrow \theta'$, with references to an

implicit program P made when necessary. The relation $\Rightarrow$ is defined inductively in terms of the link function and the relations for evaluation of expressions and predicates (whose definition we deliberately omit) by the following eleven rules.

$$\langle \varepsilon, \theta \rangle \Rightarrow \theta \qquad (R_\varepsilon)$$

$$\frac{\langle \text{state}(x) \neq \triangleright \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle \ell(P, \triangleright x)\alpha, \theta_s[x := \triangleright] \rangle \Rightarrow \theta'}{\langle (p\,?\triangleright x)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^+_{\triangleright})$$

$$\frac{\langle \text{state}(x) \neq \triangleright \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\triangleright x)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^-_{\triangleright})$$

$$\frac{\langle \text{state}(x) = \triangleright \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle \ell(P, \blacksquare x)\alpha, \theta_s[x := \blacksquare] \rangle \Rightarrow \theta'}{\langle (p\,?\blacksquare x)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^+_{\blacksquare})$$

$$\frac{\langle \text{state}(x) = \triangleright \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\blacksquare x)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^-_{\blacksquare})$$

$$\frac{\langle \text{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle \ell(P, \square x)\alpha, \theta[x := \phi] \rangle \Rightarrow \theta'}{\langle (p\,?\square x)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^+_{\square})$$

$$\frac{\langle \text{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\square x)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^-_{\square})$$

$$\frac{\langle \text{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle e, \theta \rangle \Rightarrow n \quad \langle \ell(P, \twoheadrightarrow x)\alpha, \theta_t[x \mathrel{+}= n] \rangle \Rightarrow \theta'}{\langle (p\,?\twoheadrightarrow x{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^+_{\twoheadrightarrow})$$

$$\frac{\langle \text{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\twoheadrightarrow x{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^-_{\twoheadrightarrow})$$

$$\frac{\langle \text{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle e, \theta \rangle \Rightarrow n \quad \langle \ell(P, \circ x.u)\alpha, \theta_\rho[x.u := n] \rangle \Rightarrow \theta'}{\langle (p\,?\circ x.u{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^+_{\circ})$$

$$\frac{\langle \text{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\circ x.u{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \qquad (R^-_{\circ})$$

By rule R_ε the empty sequence ε does nothing and leaves the memory unchanged.

By rule $R^+_{\triangleright}$, if the first action of the sequence is $(p\,?\triangleright x)$ and if it can be executed in state θ, i.e., if object x is in state paused or stopped and predicate p evaluates to true in θ, the configuration evaluates to the result of evaluating sequence $\ell(P, \triangleright x)\alpha$ in $\theta_s[x := \triangleright]$; otherwise, by rule $R^-_{\triangleright}$, the configuration evaluates to the result of evaluating α in θ.

Rules $R^+_{\blacksquare}$, $R^-_{\blacksquare}$, and $R^-_{\square}$ operate similarly. If the first action of the sequence can be executed, x transitions to the corresponding state and the links that depend on the action target are triggered; otherwise, the action is dropped and the next action of the sequence is considered. Rule $R^+_{\square}$ is also similar, but besides transitioning x to state stopped, it replaces the cell of x in θ by the empty cell ϕ, which resets x's state, time, and properties.

By rule $R^+_{\twoheadrightarrow}$, if the first action of the sequence is $(p\,?\twoheadrightarrow x{:}e)$ and if it can be executed in θ, x's playback time is incremented by the number to which expression e evaluates in θ, and the links of program P that depend on target $\twoheadrightarrow x$ are triggered; otherwise, by rule $R^-_{\twoheadrightarrow}$, action $(p\,?\twoheadrightarrow x{:}e)$ is dropped and the next action of the sequence is considered. By definition of memory writes, the playback time of x is reset to 0 if $\theta_t(x) + n < 0$; thus the resulting playback time is always a nonnegative integer.

By rule $R^+_{\circ}$, if the first action of the sequence is $(p\,?\circ x.u{:}e)$ and if it can be executed, property u of x is set to the number to which expression e evaluates in θ, and the links of program P that depend on target $\circ x.u$ are triggered; otherwise, by rule $R^-_{\circ}$, action $(p\,?\circ x.u{:}e)$ is dropped and the next action of the sequence is considered.

Determinism and non-termination. Theorem 1 establishes that the evaluation of action sequences is deterministic. The proof follows by induction on the structure of derivations.

Theorem 1 (Determinism). *For all* $\alpha \in \text{ActSeq}$, θ, θ_1, $\theta_2 \in \mathcal{M}$, *if* $\langle \alpha, \theta \rangle \Rightarrow \theta_1$ *and* $\langle \alpha, \theta \rangle \Rightarrow \theta_2$ *then* $\theta_1 = \theta_2$.

Theorem 2 establishes that the evaluation of action $\triangleright x$ in the empty memory Φ with $P = \triangleright x \rightarrow \square x \triangleright x$ does not converge. The proof follows by contradiction on the assumption of minimality of a hypothetical derivation of $\langle (\top \,?\triangleright x), P, \Phi \rangle \Rightarrow \theta$.

Theorem 2. *Let* $P = \triangleright x \rightarrow (\top \,?\square x)(\top \,?\triangleright x)$. *Then there is no* $\theta \in \mathcal{M}$ *such that* $\langle (\top \,?\triangleright x), P, \Phi \rangle \Rightarrow \theta$.

The above theorem implies that, under the equational semantics, the computation of reactions may not terminate in a finite number of steps, which violates the synchronous hypothesis. Similar problems occur in related languages, e.g., the problem of cyclic dependencies in SMIL's timegraph [16] (the structure used by the SMIL interpreter to control the presentation), or that of causality cycles in Esterel [2]. Here the problem is caused by infinite feedback loops in link evaluation: a link (or group of links) triggers its reevaluation endlessly. A common approach to tackle such tight loops is to impose a restriction that breaks them. For example, we could establish an upper bound to the number of times the same link or action can execute during a reaction. Though such restrictions are reasonable, we follow a more flexible path. Instead of adopting a particular a priori restriction, we introduce a linear format for programs in which links and action sequences are replaced by equivalent linear programs that always terminate.

4 The linear semantics

The abstract syntax of linear programs is mostly identical to that of equational programs presented in Section 3. The only difference is the substitution of sets ActSeq and LinkSeq by the set ActLine of linear programs defined as follows:

$$\alpha \in \text{ActLine} ::= \varepsilon \mid a[\alpha_1]\alpha_2$$

Here metavariable α is assumed to range over ActLine. Though the same metavariable is used to denote action sequences (members of ActSeq), care is taken not to mix the uses so that the correct denotation can always be inferred from the context.

The linearization procedure σ we adopt takes as input an equational program P and an action a and outputs a linear program α that represents the evaluation of a in P. The procedure σ is defined in terms of the graph of program P, which is built by interpreting its links as an adjacency list. For instance, Figure 1 depicts a Smix program and its corresponding graph. A loop in the graph indicates the possibility of a tight loop during reaction evaluation, but it does not guarantee that it will occur—its occurrence depends on the contents of the evaluation stack and media memory, both of which cannot be known statically.

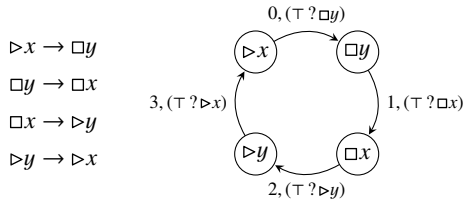


Fig. 1. A Smix program and its corresponding graph.

Given some program P and action a, procedure σ starts at the node representing the target of action a and proceeds in depth-first fashion traversing (marking) each reachable arc at most once. Its result is the linear program that implements the execution a in P. The procedure's running time is bounded to the number of arcs reachable from its point of departure; its time complexity is thus $O(n)$ where n is size of program P.

By applying σ to the program of Figure 1 with an input action $(\top\ ?\triangleright x)$, we get the linear program $\triangleright x[\Box y[\Box x[\triangleright y]]]$. This program encodes the dependencies between actions on the original equational program. To evaluate it, the kernel reads its leftmost action, $\triangleright x$, and tries to execute it. If it succeeds, in this case, if x can transition to state occurring in θ, it proceeds to evaluate the subprogram that depends on $\triangleright x$, namely, the subprogram immediately following it in square brackets, $\Box y[\Box x[\triangleright y]]$. Otherwise, it skips the brackets altogether and proceeds to evaluate the next subprogram, ε in this case. The kernel continues until there are no actions left to be executed.

Evaluation of linear programs. The evaluation of linear programs is given by the relation $\Rightarrow$ such that $\langle \alpha, \theta\rangle \Rightarrow \theta'$ iff linear program α when executed in memory θ evaluates to an updated memory θ'. Relation $\Rightarrow$ is defined inductively in terms of the relations for evaluation of expressions and predicates by the following rules. (Here we show only the rules for the evaluation of programs whose first action is a start action; the rules for the remaining actions and for the empty program are similar—they are analogous to their counterparts in the equational semantics.)

$$\frac{\langle \text{state}(x) \neq \triangleright \wedge\ p, \theta\rangle \Rightarrow \top \quad \langle \alpha_1\alpha_2, \theta_s[x := \triangleright]\rangle \Rightarrow \theta'}{\langle (p\ ?\triangleright x)[\alpha_1]\alpha_2, \theta\rangle \Rightarrow \theta'} \qquad (R^{+}_{\triangleright})$$

$$\frac{\langle \text{state}(x) \neq \triangleright \wedge\ p, \theta\rangle \Rightarrow \bot \quad \langle \alpha_2, \theta\rangle \Rightarrow \theta'}{\langle (p\ ?\triangleright x)[\alpha_1]\alpha_2, \theta\rangle \Rightarrow \theta'} \qquad (R^{-}_{\triangleright})$$

Determinism and termination. Theorem 3 establishes that the evaluation of linear programs is deterministic. The proof follows by induction on the structure of derivations.

Theorem 3 (Determinism). *For all* $\alpha \in$ ActLine*,* θ*,* θ_1*,* $\theta_2 \in \mathcal{M}$*, if* $\langle \alpha, \theta\rangle \Rightarrow \theta_1$ *and* $\langle \alpha, \theta\rangle \Rightarrow \theta_2$ *then* $\theta_1 = \theta_2$*.*

Theorem 4 establishes that the evaluation of linear programs always terminates. Its proof follows by induction on the structure of programs and depends on a lemma that establishes that $\langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta''$ iff $\langle \alpha_1, \theta\rangle \Rightarrow \theta'$ and $\langle \alpha_2, \theta'\rangle \Rightarrow \theta''$, for some θ'.

Theorem 4 (Termination). *For all* $\alpha \in$ ActLine *and* $\theta \in \mathcal{M}$*, there is a* $\theta' \in \mathcal{M}$ *such that* $\langle \alpha, \theta\rangle \Rightarrow \theta'$*.*

A consequence of Theorem 4 is the Turing-incompleteness of the computational model of linear Smix programs. One requirement for Turing-completeness is the ability to express indefinite iteration, but Theorem 4 restricts this ability, so the resulting model is not Turing-complete. This means that there are computable functions which cannot be expressed by linear Smix programs. That said, Smix's model is intentionally restricted: it aims to ease the description of interactive multimedia presentations, as opposed to the description of general algorithms. Moreover, if general computing functions are required, one can resort to external scripts, which can be embedded in the program as media objects containing Lua code.

Finally, note that the evaluation relation for linear programs determines a natural equivalence relation on ActLine: programs α_1 and α_2 are equivalent, in symbols $\alpha_1 \sim \alpha_2$, iff they evaluate to the same final memory θ' when fed with the same initial memory θ. This definition of equivalence gives rise to program reduction techniques which can be used to optimize programs. Equivalence results and the detailed proofs of the previous theorems can be found in [10].

5 Coroutine interpretation

The original implementation of Smix [10] has two parts: the language kernel, which is simply a realization of the linear semantics, and the multimedia engine, which take kernel's commands and renders the corresponding multimedia presentation. These parts are kept in isolated modules that communicate asynchronously by exchanging messages (actions). Though this design works reasonably well, an even simpler implementation is possible: we can convert (or interpret) the Smix program into a Lua script that uses the multimedia engine's synchronous API plus Lua coroutines to realize the program logic. We now describe this alternative implementation in detail.

Smix's multimedia engine code consists of a single C library, called LibPlay[4], which is built on top of GStreamer [6], a free/open-source framework for multimedia. The Lua binding of LibPlay is called LuaPlay, and has two main concepts: scene and media. A scene represents an OS-level window with audio and video output. And a media represents a media object, which is analogous to a Smix media object. The scene API consists of the following functions: (i) *new*, which creates it, (ii) *get* and *set* which manipulate its properties, (iii) *receive* which blocks awaiting for a given event, and (iv) *quit* which quits the scene. Similarly, the media API consists of the functions (i) *new* which creates a media in a given scene, (ii) *get* and *set* which manipulates the media properties, and (iii) *start*, *pause*, *stop* and *seek* which manipulates the media state and playback time.

The scene and media APIs we are considering here are synchronous: all its calls are immediately effectuated and, with exception of scene's *receive* call, execute in no (logical) time. The only call that actually "consumes" time is *receive*; in fact, it is only during this call that the engine produces audiovisual samples, and it does this until an event that matches the mask passed to *receive* is generated. Currently, LuaPlay API supports three types of events: clock ticks, user interactions (keyboard and mouse) and media object state changes.

[4] `https://github.com/TeleMidia/LibPlay`

Using the previous API, we can easily construct simple applications that wait for a single event before doing something. But as soon as we need to wait for more than one event things get complicated. There are basically two approaches to deal with the problem of awaiting on multiple events (conditions). The traditional solution is to use callbacks—we could call *receive* in a loop, passing each received event to the registered callbacks. The problem with this solution is that the program logic is "lost" in the callbacks. The alternative solution, and the one we adopt here, is to implement a parallel operation that creates new program trails dynamically. Under this approach, to wait for two events we simply create two trails and block them on the corresponding events.

In LuaPlay, this parallel operator is the scene function *par*: it creates a trail for each function received as argument and terminates the parallel composition as soon as one of them ends. In practice, we use Lua coroutines to implement the parallel composition. The *par* call creates a coroutine to represent the parallel composition of the given functions. It wraps each function into a coroutine itself and then execute these child coroutines, one at a time. If all of them yield awaiting on some condition, the composition itself yields awaiting on the combined condition. Otherwise, if one of them terminates, the composition terminates, which causes the termination of its child trails. (If *par* calls are nested, only the topmost call calls the real "await", i.e., the scene's *receive* function.)

Using LuaPlay's parallel operator *par* and an *await* operator, which is simply the coroutine yield call, we can easily implement Smix programs: the program itself is a single *par* call and each of its links is a trail that waits in a loop for the link condition (its head) and executes the corresponding linear program whenever it is awaken. Figure 2 presents the LuaPlay program that implements the example Smix program discussed at the end of Section 2. Finally, note that using this technique we can either compile Smix programs into LuaPlay programs or interpret them directly, i.e., we can write an *eval* function which takes a Smix program, builds and returns a function that is the corresponding LuaPlay program (the main trail of the *par* call).

```
1  scene:par {
2    function ()
3      while true do
4        await {type='start', media=λ}
5        exec (σ(P,▷λ))
6      end end,
7    function ()
8      while true do
9        await {type='start', media=x}
10       exec (σ(P,▷x))
11     end end,
12   function ()
13     while true do
14       await {type='start', media=y}
15       exec (σ(P,▷y))
16     end end,
17   function ()
18     while true do
19       await {type='stop', media=x}
20       exec (σ(P,□x))
21     end end
22 }
```

Fig. 2. Coroutine version of the example Smix program discussed at the end of Section 2.

6 Conclusion

In this paper, we presented the Smix language, discussed two versions its synchronous semantics, equational and linear, and proposed a novel, straightforward implementation of its linear semantics using Lua coroutines. Though we discussed most Smix features, some of them (pinned actions, limited iteration, and asynchronous actions) were deliberately omitted. These omissions, however, do not affect the formalisms and results discussed in Sections 3 and 4, nor the coroutine implementation discussed in Section 5.

We are currently investigating a continuation semantics for the coroutine interpretation of Smix programs discussed in Section 5. Our goal, in this case, is not only to relate both semantics (Smix and continuations) but also to use the continuation semantics as a basis for developing an imperative (Esterel-like) synchronous multimedia language (whose engine is LuaPlay). Besides the continuation semantics, we are also investigating approaches for verifying the behavior of Smix programs "in time", i.e., for reasoning about a sequence of chained program reactions, each describing an instant. We intend to use for this purpose Petri-PDL [11], an extension of dynamic propositional logic for Petri nets, which would allow us to model the behavior of both the language kernel and the multimedia engine, and to do so stochastically.

References

1. ABNT NBR 15606-2: Digital Terrestrial TV — Data Coding and Transmission Specification for Digital Broadcasting — Part 2: Ginga-NCL for Fixed and Mobile Receivers: XML Application Language for Application Coding. ABNT, São Paulo, SP, Brazil (2007)
2. Berry, G.: The constructive semantics of pure Esterel: Draft version 3. Tech. rep., INRIA, Sophia-Antipolis, France (2002)
3. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19(2) (1992)
4. Gaggi, O., Bossi, A.: Analysis and verification of SMIL documents. Multimedia Systems 17(6) (2011)
5. Gamatié, A.: Designing Embedded Systems with the SIGNAL Programming Language. Springer New York, New York, NY, USA (2010)
6. GStreamer Developers: GStreamer: Open source multimedia framework. `http://gstreamer.freedesktop.org`, accessed November 9, 2016
7. Ierusalimschy, R.: Programming in Lua. Lua.org, 3rd edn. (2013)
8. ITU-T Recommendation H.761: Nested Context Language (NCL) and Ginga-NCL. ITU Telecommunication Standardization Sector, Geneva, Switzerland (November 2014)
9. Kahn, G.: Natural semantics. In: STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science, 1987 Proceedings, LNCS, vol. 247 (1987)
10. Lima, G.F.: A synchronous virtual machine for multimedia presentations. Ph.D. thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil (2015)
11. Lopes, B.: Extending Propositional Dynamic Logic for Petri Nets. Ph.D. thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil (2014)
12. Picinin, D., Farines, J.M., Koliver, C.: An approach to verify live NCL applications. In: Proceedings of the 18th WebMedia, São Paulo, SP, Brazil, 15–18 October, 2012. ACM (2012)
13. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. 19, Computer Science Departement, Aarhus University, Aarhus, Denmark (1981)
14. dos Santos, J.: Multimedia Document Validation Along its Life Cycle. Ph.D. thesis, Computing Institute, UFF, Niterói, RJ, Brazil (2016)
15. dos Santos, J., Braga, C., Muchaluat-Saade, D.C.: A rewriting logic semantics for NCL. Science of Computer Programming 107–108 (2015)
16. W3C: Synchronized multimedia integration language (SMIL 3.0). Recommendation, World Wide Web Consortium (December 2008)
17. W3C: HTML5: A vocabulary and associated APIs for HTML and XHTML. Recommendation, World Wide Web Consortium (October 2014)

Interpretador e Verificador de Tipos para o Cálculo-λ Quântico com Mônadas e Setas *

José Carlos Puiati Pires[1], Eduardo Kessler Piveta[1] e Juliana Kaizer Vizzotto[1]

Universidade Federal de Santa Maria

Resumo Uma das características mais importantes da computação quântica é a superposição de estados, que pode ser interpretada como não determinismo (um efeito computacional). Uma forma de alcançar elegantemente a modelagem de efeitos computacionais em linguagens funcionais é a partir da utilização de mônadas, e através de sua forma mais genérica, as setas. Assim, o presente trabalho tem como proposta apresentar um interpretador e um verificador de tipos para o cálculo-λ quântico com a utilização de mônadas e setas.

Keywords: cálculo-λ, interpretador, computação quântica, mônadas, verificador de tipos

1 Introdução

Dentre os modelos clássicos de computação, o cálculo-λ [3] é considerado a linguagem de programação mais simples e universal. Até mesmo o conceito mais fundamental de computabilidade pode ser definido em termos do cálculo-λ. Além disso, a correspondência de Curry-Howard estabelece uma relação direta entre os termos-λ tipados e provas na lógica construtivista. Investigar essa correspondência para computação quântica tem sido uma motivação na área de linguagens de programação quântica.

Nesse contexto, o cálculo-λ quântico monádico com setas [15] é uma extensão do cálculo-λ simples com tipos, que tem como objetivo expressar programas quânticos. Entretanto, ao invés de representar diretamente combinações lineares de termos na linguagem, esse cálculo é baseado na meta-linguagem computacional introduzida por [9]. Para representar os dois tipos de computações quânticas, puras (reversíveis) e impuras (medida), utiliza-se uma construção natural chamada de setas [5, 7].

O objetivo do trabalho é propor um interpretador para o cálculo-λ quântico com a utilização de mônadas e setas, além de ser realizada a definição de um verificador de tipos para o interpretador. Como principal contribuição, espera-se ter uma linguagem de programação quântica de alto-nível executável para modelagem de algoritmos quânticos e também um estudo de propriedades da lógica quântica que fundamenta tal linguagem.

* Trabalho financiado por uma bolsa CAPES e pelo projeto STICAmSud - CAPES intitulado Foundations of Quantum Computation: Syntax and Semantics.

2 Preliminares

2.1 Computação Quântica

A unidade básica de informação clássica computável é o bit, um sistema físico clássico binário. Em computação quântica, a unidade básica de informação é representada pelo bit quântico ou qubit, um sistema físico quântico binário. O qubit é comumente representado como uma superposição de estados, através da notação *braket*[1] de Dirac [4]: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

Os valores associados às bases α e β representam as amplitudes de probabilidade relacionadas a cada base do qubit ($|0\rangle$ e $|1\rangle$). O qubit pode ser definido como um vetor em um espaço vetorial complexo (espaço de Hilbert), tal que $|\alpha|^2 + |\beta|^2 = 1$.

Formalmente, a combinação de dois ou mais estados quânticos pode ser obtida usando uma operação de produto tensorial ($\otimes$). Se $q = \alpha|0\rangle + \beta|1\rangle$ e $p = \gamma|0\rangle + \delta|1\rangle$ são dois qubits não relacionados. Ao aplicar o produto tensorial, temos $q \otimes p = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$.

O processamento de informação quântica é realizado por operadores, que possibilitam a evolução de um sistema quântico. Ele define que um sistema quântico isolado no estado $|\phi\rangle_1$ evolui para $|\phi\rangle_2$ através da aplicação de uma operação unitária $|\phi\rangle_2 = \mathbb{U}|\phi\rangle_1$. As operações quânticas tem como característica serem reversíveis, isto é, quando se conhece as operações que foram aplicadas ao qubit, é possível retornar ao seu estado inicial.

Outro tipo de operação sobre os bits quânticos é a *medição*. Ela é uma operação não reversível e pode ser explicada como uma visão clássica probabilística de um vetor de estados quânticos. Dessa forma ao aplicar a medição da base 0 em um sistema quântico da forma $\alpha|0\rangle + \beta|1\rangle$ obtemos $|0\rangle$ com $|\alpha|^2$ de probabilidade associada.

2.2 Mônadas

Com o intuito de otimizar, tornar elegante e flexibilizar a utilização de noções de computação em informática, foram apresentadas as mônadas, abordagem semântica para computações com base no modelo matemático de teoria das categorias [8]. Através da utilização das mônadas é possível encapsular as alterações de estado dos dados (efeitos colaterais) sem afetar os demais.

Moggi [8] propõe para manipular mônadas, utilizar os elementos da tripla Kleisli (M,η,$\gg=$):

i. M, construtor do tipo monádico;
ii. η, uma adaptação da função identidade $id\ x = x$, representada como $return :: a \rightarrow Ma$;
iii. $\gg=$, uma adaptação da composição de funções $(f\ .\ g)\ x = f\ (g\ x)$, representada como $\gg= :: Ma \rightarrow (a \rightarrow Mb) \rightarrow Mb$.

[1] O nome *braket* surge através de uma convenção em que um vetor de coluna é chamado "ket" e sua notação é demonstrada por $|\ \rangle$ e um vetor de linhas é chamado "bra" e tem como notação $\langle\ |$.

2.3 Setas

As mônadas demonstram um grande poder computacional para várias operações, porém em alguns casos elas não podem ser utilizadas. Mônadas não são aplicadas à funções que recebem múltiplas entradas, especialmente quando algumas entradas são estáticas e outras dinâmicas. Assim, como um modelo mais genérico de mônadas, foram propostas as setas (*arrows*) [5].

A utilização de mônadas e setas é diferente para as representações de estado, embora ambas trabalhem para sequencializar efeitos computacionais. As mônadas servem mais para controle do fluxo global, como exceções. As setas são mais indicados para modelar propriedades de transformações de fluxos, como estruturar circuitos computacionais. [2]

As setas são um modelo mais genérico para sequencializar computações, e podem ser vistas como uma generalização de mônadas [6].

3 Modelando Efeitos Quânticos com Mônadas

Bits quânticos, pela sua características de superposição de estados, podem ser considerados como elementos com efeitos computacionais. Essas estruturas que apresentam não determinismo podem ser modeladas com a utilização de mônadas [10]. Assim, temos o não determinismo definido, através da tripla Kleisli (T,η,$\gg=$), como $TA = \mathcal{P}(A)$, sendo $\mathcal{P}(A)$ o conjunto de valores possíveis para A.

Através da proposta monádica para tratar o não determinismo, podemos criar estados quânticos e operações sobre eles [14]. Pelas bases do conjunto A e $\gg=$ temos o significado de linearidade. Podemos descrever o bit quântico como um vetor no espaço bidimensional complexo. Considerando as bases $A = \{0, 1\}$, e os números complexos $\mathbb{C} = \{\alpha, \beta\}$, temos $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. Assim, a função $\gg=$ deve fazer um mapeamento das possibilidades de ocorrência de cada base e relacionar a sua amplitude de probabilidade, na forma $\mathcal{P}(A \times \mathbb{C})$.

Com intuito de representar valores quânticos, foi proposta uma abordagem de computação quântica utilizando mônadas, com seus valores de acordo com a tripla Kleisli:

i. M, Vec a = a → PA como o construtor do tipo (Vec a), que mapeia uma base a para uma amplitude de probabilidade P A;
ii. η, return :: a → Vec a, transforma o valor em um dado monádico;
iii. $\gg=$, bind :: Vec a → (a → Vec b) → Vec b, sequencializa uma computação de Vec a para Vec b

4 Cálculo-λ com *Double Effect*

Não apenas na forma de função as mônadas e setas podem ser úteis para representar a computação quântica. Os efeitos colaterais podem ser representados em termos do cálculo-λ. Várias ideias foram propostas nesse contexto [11,12], além de também serem definidas regras de tipos para interpretação da computação quântica [13].

A partir desses estudos foram propostas variações do cálculo-λ quântico para que possam trabalhar com a informação quântica utilizando mônadas e setas [15], uma extensão do cálculo-λ simplesmente tipado para expressar algoritmos quânticos.

Em computação quântica existem dois tipos de estado: (i) estados puros, a esses é possível aplicar operações lineares e unitárias, bem como estabeler processos de reversibilidade e (ii) estados impuros, resultantes de uma medição. Para representar os dois tipos de estados quânticos podem ser utilizadas setas [5,7].

4.1 Cálculo-λ Quântico Monádico

Formalmente a extensão do cálculo-λ simplesmente tipado para o cálculo-λ monádico necessita de duas operações, a primeira para transformar valores puros em cálculos monádicos, $[M]_M$, e a segunda para compor uma sequência de efeitos, let_M, essas duas operações são análogas à return e bind [10].

Na Figura 1 podemos visualizar as regras de tipos para o calculo-λ quântico bem como a terminologia utilizada para representar operadores quânticos. E na Figura 2 é possível visualizar as regras de avaliação utilizadas neste cálculo [15].

Figura 1. Sintaxe e Regras de Tipo do Cálculo-λ Monádico

Sintaxe

Amplitudes de Probabilidade	$\alpha, \beta \in \mathbb{C}$
Definição de Tipos	$\mathbf{Vec}\ A \quad = \quad A \to \mathbb{C}$
Tipos	$A, B, C \ ::= \ldots \mid \mathbf{Vec}\ A$
Termos	$L, M, N \ ::= \ldots \mid [M]_M \mid \mathsf{let}_M\ x = M\ \mathsf{in}\ N \mid \mathsf{vzero} \mid + \mid -$

Tipo Monádico

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash [M]_M : \mathbf{Vec}\ A}$$

Tipo Let

$$\frac{\Gamma \vdash M : \mathbf{Vec}\ A \quad \Gamma, x : A \vdash N : \mathbf{Vec}\ B}{\Gamma \vdash \mathsf{let}_M\ x = M\ \mathsf{in}\ N : \mathbf{Vec}\ B}$$

Tipo Mônada Plus

$$\frac{\Gamma \vdash M, N : \mathbf{Vec}\ A}{\Gamma \vdash M + N : \mathbf{Vec}\ A}$$

Tipo Mônada Zero

$$\frac{}{\Gamma \vdash \mathsf{vzero} : \mathbf{Vec}\ A}$$

Tipo Mônada Minus

$$\frac{\Gamma \vdash M, N : \mathbf{Vec}\ A}{\Gamma \vdash M - N : \mathbf{Vec}\ A}$$

Tipo Produto Escalar

$$\frac{\Gamma \vdash M : \mathbf{Vec}\ A}{\Gamma \vdash \alpha * M : \mathbf{Vec}\ A}$$

Figura 2. Regras de Avaliação do Cálculo-λ Monádico

Regras para o Let$_M$

$(left_M)$	$\mathsf{let}_M\ x = [L]_M\ \mathsf{in}\ N$	$= N[x := L]$
$(right_M)$	$\mathsf{let}_M\ x = L\ \mathsf{in}\ [x]$	$= L$
$(assoc_M)$	$\mathsf{let}_M\ y = (\mathsf{let}_M\ x = L\ \mathsf{in}\ N)\ \mathsf{in}\ T$	$= \mathsf{let}_M\ x = L\ \mathsf{in}\ (\mathsf{let}_M\ y = N\ \mathsf{in}\ T)$

Regras para a Mônada plus

$(vzero_+)$	$\mathsf{vzero} + a$	$= a$
$(vzero_+)$	$a + \mathsf{vzero}$	$= a$
$(assoc_+)$	$a + (b + c)$	$= (a + b) + c$
$(left_M^{vzero})$	$\mathsf{let}_M\ x = \mathsf{vzero}\ \mathsf{in}\ T$	$= \mathsf{vzero}$
$(left_M^+)$	$\mathsf{let}_M\ x = (M + N)\ \mathsf{in}\ T$	$= (\mathsf{let}_M\ x = M\ \mathsf{in}\ T) + (\mathsf{let}_M\ x = N\ \mathsf{in}\ T)$

A partir da definição do calculo-λ quântico é possível modelar todos estados quânticos puros, além de realizar computações quânticas reversíveis, ou seja, aplicar operações lineares.

Apesar de ser possível representar os estados puros, deseja-se também a representação de estados quânticos impuros, i.e., estados obtidos após a medição. Para utilizar a medição no cálculo pode ser utilizada a generalização de mônadas, chamada setas [15].

4.2 Cálculo-λ Quântico com Setas

Apenas com operações reversíveis e estados puros não é possível apresentar a medição. Entretanto, ao utilizar matrizes de densidade é possível modelar formas mais generalizadas de estados quânticos, e representar ambos estados, puros e mistos. Através de superoperadores é possível se obter uma generalização de computações quânticas e representar operações lineares (reversíveis) e a medida [14].

Com base no *core* do cálculo com setas [7] foi proposto um cálculo-λ quântico utilizando-se de mônadas e setas. A sintaxe do cálculo é apresentada na Figura 3.

Ao cálculo-λ simplesmente tipado com mônadas, foram adicionadas novas definições. Uma característica importante do cálculo com setas é que o domínio para efeitos colaterais deve ter dois contextos diferentes: Γ para as variáveis derivadas de abstrações-λ ordinárias e Δ variáveis derivadas de abstrações de setas [15].

5 Interpretador para o Cálculo Quântico com Mônadas e Setas

O interpretador e verificador de tipos propostos vêm sendo desenvolvidos na linguagem funcional Haskell. A partir do *core* do cálculo-λ simplesmente tipado,

Figura 3. Cálculo-λ Quântico com Setas

Sintaxe

$Typedef$	$\mathbf{Dens}\ A$	$= (A, A) \to \mathbb{C}$
$Typedef$	$\mathbf{Super}\ A\ B$	$= (A, A) \to \mathbf{Dens}\ B$
$Types$	A, B, C	$::= \ldots \mid \mathbf{Dens}\ A \mid \mathbf{Super}\ A\ B$
$Terms$	L, M, N	$::= \ldots \mid \lambda^{\bullet} x.Q$
$Commands$	P, Q, R	$::= L \bullet M \mid [M]_A \mid \mathsf{let}_A\ x = P\ \mathsf{in}\ Q$
$Environments$	Γ, Δ	$::= x_1 : A_1, \ldots, x_n : A_n$

Tipo Arrow

$$\frac{\Gamma; x : A \vdash Q\ !\ \mathbf{Dens}\ B}{\Gamma \vdash \lambda^{\bullet} x.Q : \mathbf{Super}\ A\ B} \qquad \frac{\Gamma \vdash L : \mathbf{Super}\ A\ B \quad \Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash L \bullet M\ !\ \mathbf{Dens}\ B}$$

$$\frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash [M]_A\ !\ \mathbf{Dens}\ A} \qquad \frac{\Gamma; \Delta \vdash P\ !\ \mathbf{Dens}\ A \quad \Gamma; \Delta, x : A \vdash Q\ !\ \mathbf{Dens}\ B}{\Gamma; \Delta \vdash \mathsf{let}_A\ x = P\ \mathsf{in}\ Q\ !\ \mathbf{Dens}\ B}$$

Regras de Avaliação

$(\beta^{\rightsquigarrow})$	$(\lambda^{\bullet} x.Q) \bullet M$	$= Q[x := M]$
$(\eta^{\rightsquigarrow})$	$\lambda^{\bullet} x.(L \bullet [x]_A)$	$= L$
$(left)$	$\mathsf{let}_A\ x = [M]_A\ \mathsf{in}\ Q$	$= Q[x := M]$
$(right)$	$\mathsf{let}_A\ x = P\ \mathsf{in}\ [x]$	$= P$
$(assoc)$	$\mathsf{let}_A\ y = (\mathsf{let}\ x = P\ \mathsf{in}\ Q)\ \mathsf{in}\ R$	$= \mathsf{let}_A\ x = P\ \mathsf{in}\ (\mathsf{let}_A\ y = Q\ \mathsf{in}\ R)$

foram sendo acrescidos elementos que permitam o desenvolvimento de estruturas capazes de representar dados e realizar operações quânticas.

A semântica operacional utilizada nesse cálculo foi apresentada nas seções anteriores pela Figuras 2 e 3, assim como o sistema de tipos, através Figuras 1 e 3. A partir delas foi possível definir a sintaxe abstrata dos termos do cálculo-λ.

O interpretador está sendo desenvolvido com a notação *nameless*, que modifica os termos do cálculo para termos sem nome. Esta notação tem como objetivo de evitar, ao realizar a substituição, a captura de variáveis. Ela utiliza índices (índices de Bruijn) em vez de variáveis, prevenindo de antemão a captura de variáveis [1].

5.1 Cálculo-λ Quântico Monádico

Como visto na Seção 4.1, dados monádicos podem ser utilizados para representar bits quânticos e operações reversíveis. Para que isso seja obtido, primeiramente foram adicionados alguns elementos sintáticos, presentes na Figura 1. Definidos pelo construtor de tipos **data** temos o tipo **TLam** que representa os termos lambda do interpretador, assim adicionamos os termos monádicos:

```
data TLam = ...
          | TMon TLam
          | TLetM Char TLam TLam
          | TVZero
          | TMPlus TLam TLam
          | TMMinus TLam TLam
          | TMScalar Double TLam
```

Ao definir a árvore de sintaxe abstrata para esses termos temos: em **TMon** a representação um constrututor de uma mônada; com a proposta de sequencializar computações, o termo **TLetM**; o **TVZero** como um numeral monádico; para representar a superposição dos bits quânticos foram construidos **TPlus** e **TMMinus**; e por fim **TMScalar**, que associa um amplitude de probabilidade a uma base (para o interpretador foi utilizado **TTrue** para $|1\rangle$ e **TFalse** para $|0\rangle$).

Através da sintaxe abstrata é possível representar os bits quânticos $|1\rangle$ e $|0\rangle$, bem como a superposição entre eles $|0\rangle + |1\rangle$:

$|1\rangle$ = TMPlus (TMScalar 1.0 (TMon TTrue)) (TMScalar 0.0 (TMon TFalse))

$|0\rangle$ = TMPlus (TMScalar 0.0 (TMon TTrue)) (TMScalar 1.0 (TMon TFalse))

$|0\rangle+|1\rangle$ = TMPlus (TMScalar $(1/\sqrt{2})$ (TMon TTrue)) (TMScalar $(1/\sqrt{2})$ (TMon TFalse))

Ao verificador de tipos foi acrescido o tipo **TypeVecA**, que representa um vetor de tipos sobre uma base computacional A. Primeiramente foi definido o sinônimo de tipos **TypeContext**, como o contexto de tipos a ser verificado pela função **TypeOf** (ambos vistos abaixo), que ao receber um contexto de tipos e um termo, retorna o tipo desse termo.

```
type TypeContext = [(Char,Type)]

typeOf :: TypeContext -> TLam -> Type
```

Com a semântica operacional definida, foi construida a avaliação dos termos sem nome através da chamada por valor (*call by value*), definida pela função **interpretNLam**:

```
interpretNLam :: NLam -> NLam
interpretNLam t = let t' = evalCBVNL t
                  in if t' == t then t'
                     else interpretNLam t'
```

através dela é possível avalialiar recursivamente o termo sem nome. A função **evalCBVNL** realiza um passo de avaliação, ela é acionada até que o termo esteja na sua forma normal.

A partir do verificador de tipos e das regras de avaliação é possível sequencializar as duas operações através da função **interpret**

```
interpret :: TLam -> TLam
interpret t = if (isWellTyped (typeOf contextType t))
                 then let t' = removeNames context t
```

```
            in restoreNames context (interpretNLam t')
        else error "Erro na validacao de tipos"
```

que possibilita a união das duas funções. Primeiramente é verificada a tipagem do termo, no caso de ser bem tipado inicia o processo de avaliação. A função **removeNames** realiza a renomeiação dos nomes em índices de Bruijn, então os termos são avaliados pela função **interpretNLam**, e à forma normal desse termo é aplicada a função **restoreNames** que faz o papel inverso ao **removeNames**, ou seja, restaura os nomes aos termos.

Para exemplificar a utilização do interpretador, podemos utilizar a operação de hadamard, uma operação unitária aplicada a um qubit que estabelece a superposição de estados do qubit. Ela pode ser interpretada como uma abstração que recebe um dado do tipo **TypeBool** e conforme o valor aplicado, retorna $|1\rangle$ ou $|0\rangle$, definidos anteriormente.

```
hadamard :: TLam
hadamard = Abs 'x' TypeBool (TIf (Var 'x') |1⟩ |0⟩)
```

5.2 Cálculo-λ Quântico com Setas

Para serem efetuadas computações de *double-effect* (realizar operações sobre estados quânticos puros e impuros) são necessários adicionar novos termos de setas ao interpretador. Dessa forma foram adicionados os termos de setas:

```
data TLam = ...
          | TArrow TLam
          | TAAbs Char Type TLam
          | TAApp TLam TLam
          | TALet Char TLam TLam
          | TAPlus TLam TLam
          | TAMinus TLam TLam
```

Ao definir a árvore de sintaxe abstrata para esses termos temos: **TArrow** como o construtor de uma seta que representa uma matriz de densidade; assim podemos definir funções que transformam matrizes de densidade em novas matrizes de densidade, superoperadores, com a contrução da abstração para setas **TAAbs**; o termo **TAApp** é utilizado para representar a aplicação de um superoperador em uma matriz de densidade; **TALet** representa a composição de funções de setas, bem como a composição monádica; para representar a superposição dos termos de setas foram incluidos **TAPlus** e **TAMinus**.

A organização dos contextos foi modificada, tal que, para o cálculo com setas são necessário dois contextos, foi incluído o novo contexto Δ que armazena as variáveis derivadas de abstrações de setas, dessa forma o contexto de tipos é representado através de uma dupla de listas de tuplas, em que a primeira dupla representa o contexto Γ e a segunda dupla o contexto Δ

```
type TypeContext = ([(Char,Type)],[(Char,Type)]).
```

Também foram incluídos novos tipos ao interpretador a partir das regras da Figura 3: **TypeComm Type** para definição de um tipo comando de efeitos colaterais que recebe um **TypeDens**; **TypeDens Type** representando uma matriz de densidade e por fim **TypeSuper Type Type** que define um tipo de um superoperador, que recebe um tipo e deriva para um novo tipo.

```
typeOf ctx (TALet x t1 t2) =
        let tyT1 = typeOf ctx t1
            tyT1' = removeFromArrow (tyT1)
            ctx' = addArrowType ctx x tyT1'
            tyT2 = typeOf ctx' t2
        in if (isNotError tyT1')
           then if (isNotError tyT2)
            then if (isCommand tyT2)
                 then tyT2
                 else TypeErr "O segundo termo deve ser
                 do tipo Command"
            else tyT2 --Retorna o erro
      else tyT1' --Retorna o erro
```

Por *pattern matching* (casamento de padrões), a função encontra o termo a ser avaliado pela função **TypeOf**, no caso acima temos a verificação do tipo para o termo **TALet**, a abstração de setas, que recebe uma variável (**x**) e dois termos (**t1** e **t2**). Primeiramente é removido o tipo de **t1** e aciona a função **removeFromArrow** que verifica se o primeiro termo é **TypeComm** e retira-o de um tipo de setas, o termo com efeitos de setas é adicionado ao contexto de setas Δ pela função **addArrowType**, em um contexto diferente das variáveis derivadas de abstrações-λ ordinárias. Prosseguindo na função principal, o tipo de **t2** é removido e são iniciadas as verificações se os tipos derivados de **t1** e **t2** são erros ou não são do tipo **TypeComm**.

6 Conclusão

Foi apresentado nesse trabalho o desenvolvimento de um interpretador e verificador de tipos para o cálculo-λ quântico com utilização de mônadas e setas.

Apesar de ser possível realizar e entender vários aspectos da computação quântica utilizando o interpretador e o verificador de tipos, os mesmos não estão completos. Até o momento, não é possível realizar operações de medida que utilizam estados quânticos mistos, embora tenham sido representados alguns termos e tipos que definem o cálculo com setas.

Para etapas futuras do trabalho, estão previstos a inclusão de termos e tipos para manipular a medida e também a definição de um sistema de tipos para controlar operações quânticas reversíveis.

Referências

1. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) 75(5), 381 – 392 (1972), http://www.sciencedirect.com/science/article/pii/1385725872900340
2. Capretta, V., McBride, C., Lindley, S., Wadler, P., Yallop, J.: Proceedings of the second workshop on mathematically structured functional programming (msfp 2008) idioms are oblivious, arrows are meticulous, monads are promiscuous. Electronic Notes in Theoretical Computer Science 229(5), 97 – 117 (2011), http://www.sciencedirect.com/science/article/pii/S1571066111000557
3. Church, A.: An unsolvable problem of elementary number theory. American Journal of Mathematics 58(2), 345–363 (1936)
4. Dirac, P.: The Principles of Quantum Mechanic. Clarendon Press; 3rd edition (April 1947)
5. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37, 67–111 (May 2000)
6. Hughes, J.: Programming with Arrows, pp. 73–129. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11546382_2
7. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus. Journal of Functional Programming pp. 51–69 (2010)
8. Moggi, E.: An abstract view of programming languages. Edinburgh University (1989)
9. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of LICS-1989. pp. 14–23. IEEE Computer Society (1989)
10. Mu, S.C., Bird, R.: Functional quantum programming. In: Asian Workshop on Programming Languages and Systems. KAIST, Dajeaon, Korea (dec 2001), http://www.cs.ox.ac.uk/people/richard.bird/online/MuBird2001Functional.pdf
11. Selinger, P.: Towards a quantum programming language. Mathematical. Structures in Comp. Sci. 14(4), 527–586 (Aug 2004), http://dx.doi.org/10.1017/S0960129504004256
12. Selinger, P., Valiron, B.: A Lambda Calculus for Quantum Computation with Classical Control, pp. 354–368. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11417170_26
13. Selinger, P., Valiron, B.: Quantum lambda calculus. In: Gay, S., Mackie, I. (eds.) Semantic Techniques in Quantum Computation. pp. 135–172. Cambridge Univ. Press (2009)
14. Vizzotto, J.K., Altenkirch, T., Sabry, A.: Structuring quantum effects: superoperators as arrows. CoRR abs/quant-ph/0501151 (2005), http://dblp.uni-trier.de/db/journals/corr/corr0501.html#abs-quant-ph-0501151
15. Vizzotto, J.K., Calegaro, B.C., Piveta, E.K.: A double effect lambda-calculus for quantum computation. In: Proceedings of SBLP-2013. LNCS, vol. 8129, pp. 61–74 (2013)
16. Vizzotto, J.K., Du Bois, A.R., Sabry, A.: The Arrow Calculus as a Quantum Programming Language, pp. 379–393. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02261-6_30

Formalization of the Undecidability of the Halting Problem for a Turing Complete Functional Language

Thiago Mendonça Ferreira Ramos and Mauricio Ayala-Rincón*

Universidade de Brasília, Brasília, DF , Brazil,
thiagomendoncaferreiraramos@yahoo.com.br, ayala@unb.br

Abstract. Checking whether a program is or not terminating is undecidable. Despite this, there exist several techniques to check termination of specifications and programs, that are of great relevance for the automation of termination in proof assistants and for the improvement of the compilers of future programming languages. The minimal first-order functional language `PVS0` with its operational semantics is part of the developments PVS0 and CCG, formalized in the proof assistant PVS. These developments included the specification of notions of semantic termination and *Turing's ranking function* termination as well as two *size change termination* based technologies: *calling context graphs* and *matrix weighted graphs*. The developments include formalizations that these four technologies for checking termination are equivalent. In this paper we present a PVS formalization of the Turing completeness for the `PVS0` language as well as a direct formalization of the undecidability of the halting problem for this language.

1 Introduction

Checking termination of programs is fundamental in computer science because the correctness of procedures depends on termination of processes. An example is the problem of threads termination of keyboard drivers in operational systems [2]. If after pressing a key a thread does not stop, the keyboard control is lost. Determining whether a program stops is an undecidable problem that is related with the halting problem [7] and, therefore it is not possible to construct a compiler that for all possible inputs (programs) verifies termination.

Despite this, it is possible to construct semi-decision algorithms that, given a program, can correctly answer "stops" or "unknown". In order to do this, several approaches have been developed which use equivalent notions of termination. The most updated progress in this area is regularly reported in the *Annual International Workshop on Termination and Termination Competition*[1].

For each different definition of termination, techniques to check termination might be developed [3]. Among these techniques, we emphasize these based on

* Authors respectively supported by CNPq MSc shoolarship and research grant.

[1] `http://www.termination-portal.org`

the *size-change termination principle* [5] such as *Calling Context Graphs* [6] and *Matrix Weighted Graphs* [1], that are used for functional specifications, and these based on the construction of *reduction* and *simplification orderings* for term rewriting systems as well as *Dependency Pairs* in [4] . All these techniques are related with each other since term rewriting is the formal framework for reasoning about functional programming.

PVS0 and CCG are PVS developments available in the NASA LaRC PVS library (at https://github.com/nasa/pvslib), including more than 400 lemmas, that were formalized by members of the NASA LaRC formal methods and our group of Theory of Computation in Brasília. In these developments the minimal functional language `PVS0` is specified. `PVS0` is called "minimal" because it contains just the necessary grammatical elements to be Turing complete. PVS0 and CCG include the formalization of notions such as semantic evaluation and the notion of semantic termination for programs written in `PVS0`. Additionally, they include formalizations of other notions of termination such as Turing's ranking functions, as well as matrix weighted graph and calling context graph. The main formalizations in these developments are related with theorems of equivalence between these different termination technologies. This is of great relevance for increasing the power of automation of proof assistants such as PVS providing different semantics of termination as well as the associated mechanisms to guide the construction of inductive proof schemas based on these technologies.

In this work we will only use the notions of semantic termination and ranking function termination. Ranking function termination corresponds to the semantics of termination in PVS that is implemented through a static analysis of recursive functions in which, according to a measure function on the parameters, termination "Type Correctness Conditions" (termination TCCs for short) are built that express decreasement of the parameters after each recursive call.

The main contributions of this work are:

- a formalization of the Turing completeness of the `PVS0` language.
- a direct formalization of the undecidability of the halting problem for `PVS0`.

The former implies that `PVS0` has the necessary expressiveness for computing all "computable" functions (i.e., partial recursive functions). The latter means that there exists no `PVS0` program that having as input the encoding of a `PVS0` program decides whether that program halts for all possible inputs. Indeed, the former gives as corollary the second, but the formalization of the undecidability of the halting problem is developed explicitly.

2 Definitions of Semantic and TCC Termination

The grammar of the language `PVS0` is given as:

$$\mathcal{E}xpr ::= rec(\mathcal{E}xpr)|op1(\mathcal{E}xpr)|op2(\mathcal{E}xpr,\mathcal{E}xpr)|ite(\mathcal{E}xpr,\mathcal{E}xpr,\mathcal{E}xpr)|vr|cnst$$

Above, rec is the symbol for recursion, $op1$ and $op2$ are symbols for unary and binary operators, ite is the symbol for the branching operator (IF THEN ELSE), and vr and $cnst$ for variable and constant.

We assume non-empty disjoint sets for the constant symbols $CONST$, variables VR, binary operators $OP2$ and unary operators $OP1$. The set $\mathcal{V}al$ is the set of inputs as well as outputs of the language. The semantic evaluation is given from an interpretation of constant, variable, unary and binary operator symbols, through the mapping:

$$\mathfrak{I} : (CONST \to \mathcal{V}al) \cup (OP1 \to \mathcal{V}al \to \mathcal{V}al) \cup (OP2 \to (\mathcal{V}al \times \mathcal{V}al) \to \mathcal{V}al) :$$

- If $c \in CONST$, then $c^{\mathfrak{I}} \in \mathcal{V}al$ is the value mapped from c.
- If $g \in OP1$, then $g^{\mathfrak{I}} \in \mathcal{V}al \to \mathcal{V}al$ is the unary function mapped from g.
- If $h \in OP2$, then $h^{\mathfrak{I}} \in (\mathcal{V}al \times \mathcal{V}al) \to \mathcal{V}al$ is the binary function mapped from h.

The semantic evaluation of an expression is given by the relation $\varepsilon : \mathcal{E}xpr \times \mathcal{E}xpr \times \mathcal{V}al \times \mathcal{V}al \to \texttt{bool}$:

$$\begin{aligned}
\varepsilon(e, e_f, \beta, \nu) := \texttt{CASES } e \texttt{ OF}\\
cnst &: \nu = cnst^{\mathfrak{I}};\\
vr &: \nu = \beta;\\
op1(e_1) &: \exists \nu_1 \in \mathcal{V}al : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge\\
&\quad \nu = op1^{\mathfrak{I}}(\nu_1);\\
op2(e_1, e_2) &: \exists \nu_1, \nu_2 \in \mathcal{V}al : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \varepsilon(e_2, e_f, \beta, \nu_2) \wedge\\
&\quad \nu = op2^{\mathfrak{I}}(\nu_1, \nu_2);\\
ite(e_1, e_2, e_3) &: \exists \nu_1 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge\\
&\quad \texttt{IF } \nu_1 \texttt{ THEN } \varepsilon(e_2, e_f, \beta, \nu) \texttt{ ELSE } \varepsilon(e_3, e_f, \beta, \nu);\\
rec(e_1) &: \exists \beta' \in \mathcal{V}al : \varepsilon(e_1, e_f, \beta, \beta') \wedge\\
&\quad \varepsilon(e_f, e_f, \beta', \nu)
\end{aligned}$$

Where $op1^{\mathfrak{I}}$ and $op2^{\mathfrak{I}}$ are total functions.

Semantic evaluation performs using lazy evaluation, that is, expressions are evaluated only if necessary. The relation $\varepsilon(e, e_f, \beta, \nu)$ means that the evaluation of the expression e regarding the recursive expression e_f and with input β produces as output ν. But if the recursion does not stop, the predicate is false.

The predicate for semantic termination of the expression $T_\varepsilon : \mathcal{E}xp \to \texttt{bool}$ is defined as follows:

$$T_\varepsilon(e_f) := \forall\ \beta,\ \exists\ \nu \in \mathcal{V}al :\ \varepsilon(e_f, e_f, \beta, \nu)$$

It expresses that for all inputs the expression e_f must produce an output in order to be considered terminating.

As an example consider the greatest common divisor. Initially, suppose that the following functions are implemented:

$$\begin{aligned}
Z^{\mathfrak{I}}_{ero}(m, n) &:= m = 0 \vee n = 0 & \qquad S^{\mathfrak{I}}_{um}(m, n) &:= (m + n, 0)\\
G^{\mathfrak{I}}_{eq}(m, n) &:= n \geq m & \qquad S^{\mathfrak{I}}_{ub}(m, n) &:= (m, n - m)\\
P^{\mathfrak{I}}_{er}(m, n) &:= (n, m)
\end{aligned}$$

Then, the function $gcd : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, can be specified in PVS0 as below:

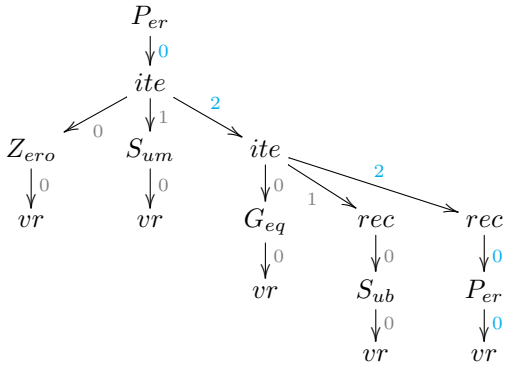
$$gcd := ite(Z_{ero}(vr), S_{um}(vr), ite(G_{eq}(vr), rec(S_{ub}(vr)), rec(P_{er}(vr))))$$

Where the answer is given as the first element of the output tuple. Above, if one element of the tuple is equal to zero, *gcd* is the sum of the elements. If the second one greater than or equal to the first one, *gcd* is recursively called with the first element and the second minus the first one. Otherwise *gcd* is recursively called with the swapped tuple.

Another definition for termination checks that there exists a kind of "decrement" by a well-founded relation between the parameters of the function (the formal parameters) and the corresponding arguments of the recursive calls (the actual parameters), assuming that the conditions to execute the recursive call hold. This notion is adopted as the semantics of termination in several proof assistants and is known as the *ranking function* technique ([8]) and it is formalized to be equivalent to the previous given notion of termination. This notion is implemented in PVS through the construction of the so called termination TCCs (Type Correctness Conditions) that uses a measure function and a well-founded relation provided by the specifier. Essentially, it is required that the measure of the formal parameters be greater than the measure of the actual parameters of all possible recursive calls. For instance *gcd* above, might use the measure function $lex2((m, n)) := \omega m + n$, and apply the ordering $(>)$ over ordinals. This corresponds to the lexicographic ordering over pairs of naturals. Since this is a well-founded order, it is easy to check that the required decreasement conditions over parameters and actuals of the recursive calls hold, that is:

- if $n \geq m$, then $lex2((m, n)) > lex2((m, n - m))$
- if $n < m$, then $lex2((m, n)) > lex2((n, m))$

Expressions are represented by their syntactic trees. This is used to define how to determine the subexpression given by a path in the execution tree, which paths are valid and stock the boolean expressions that execute a recursive call. Paths are sequences of naturals ($[\mathbb{N}]$) that represent subexpressions, when reading them backwards. Paths might be extended putting naturals in front of the sequences. Below, $[T]$ means a list of type T. The examples of paths in the expression $P_{er}(gcd)$ are $[1, 0]$, $[0, 1, 0]$ and $[0, 0, 2, 2, 0]$ that are highlighted below.



It is possible to build the chain of conditions in an execution path. A data structure is needed to store whether the condition corresponds to the THEN or ELSE branch of a branching instruction in the execution path. To that effect, one may use the following abstract data structure which type is $\mathcal{E}xprbool$:

$exprbool : \mathcal{E}xpr \to \mathcal{E}xprbool \qquad exprnot : \mathcal{E}xpr \to \mathcal{E}xprbool.$

If the extraction of subexpression is from THEN, then the condition is stored using $exprbool$, otherwise, it is stored by $exprnot$.

A function that extracts the chain of path conditions in recursive definitions was specified, here denoted as Π. These are examples of extracted path conditions for the expression $P_{er}(gcd)$:

- $\Pi(P_{er}(gcd), [1, 0]) = [exprbool(Z_{zero}(vr))]$
- $\Pi(P_{er}(gcd), [0, 1, 0]) = [exprbool(Z_{zero}(vr))]$
- $\Pi(P_{er}(gcd), [2, 2, 0]) = [exprnot(G_{eq}(vr)), exprnot(Z_{zero}(vr))]$

Calling contexts are registers that contain information about a recursive call in an expression e and that consist of the expression of the recursive call itself, the path conditions and the execution path until the recursive call.

Let $cc := \langle rec\_expr : \{a : \mathcal{E}xpr | a = rec(b)\},\ cnds : [\mathcal{E}xprbool],\ path : [\mathbb{N}]\rangle$ be a calling context of the expression e. It is a valid calling context of e if rec_expr is the subexpression of e in $path$ and $cnds$ are the related path conditions. The predicate $Valid\_cc : \mathcal{E}xpr \to \mathcal{E}xpr\_cc \to \texttt{bool}$ expresses this. $Valid\_cc(e)(cc)$ or $cc : Valid\_cc(e)$ express that cc is a valid calling context for the expression e.

Above, the access of the elements of the data structure is given by the primitives: rec_expr to the recursion, $cnds$ to the list of conditions and $path$ to the valid path until the recursive call.

In order to evaluate the path conditions, one applies the function $EC : \mathcal{E}xpr \times [\mathcal{E}xprbool] \times \mathcal{V}al \to \texttt{bool}$, where $EC(e, cnds, \beta)$ evaluates the conditions $cnds$ with β as an input argument considering the expression e as the recursive function.

$$
\begin{aligned}
EC(e, cnds, \beta) := \texttt{CASES}\ cnds\ \texttt{OF}& \\
[\,] &: \texttt{true}; \\
[a : l] &: (\texttt{CASES}\ a\ \texttt{OF} \\
&\quad exprbool(c) : \varepsilon(c, e, \beta, \nu)\ \ \wedge\ \ \nu; \\
&\quad exprnot(c) : \varepsilon(c, e, \beta, \nu)\ \ \wedge\ \ \texttt{NOT}\ \nu)\ \ \wedge\ \ EC(e, l, \beta)
\end{aligned}
$$

To evaluate termination by ranking function, it is necessary to define a measure mapping $\mu : \mathcal{V}al \to \mathcal{MT}$ where $\mathcal{MT}$ is a metric space with a well-founded order $\prec$. Well foundedness is specified as usual: $\forall\ P : \mathcal{MT} \to \texttt{bool} : ((\exists\ e : P) \Rightarrow \exists\ (m : P),\ \forall (a : P) : \neg(a \prec m))$. Thus, the definition of termination by TCC of an expression e_f is specified as:

$$
\begin{aligned}
T_\zeta(e_f) := \exists\ \mu,\ \forall\ (\beta,\ cc : Valid\_cc(e_f),\ \nu) :& \\
\varepsilon(get\_arg(rec\_expr(cc)), e_f, \beta, \nu) \wedge EC(e_f, cnds(cc), \beta) \Rightarrow \mu(\nu) \prec \mu(\beta)&
\end{aligned}
$$

Above, get_arg get the argument of the recursive expression. In PVS, the termination above is called TCC termination. PVS builds TCC termination (and type) obligations that should be proved in order to guarantee well-definedness of the specified functions.

3 Undecidability of the Halting Problem for PVS0

The equivalence between semantic termination and TCC termination was formalized in the PVS0 development: $\forall e_f : \mathcal{E}xpr\ T_\varepsilon(e_f) \equiv T_\zeta(e_f)$. The undecidability of the halting problem (next lemma) is formalized using these notions of termination over PVS0 and the technique of diagonalization of Cantor.

Lemma 1 (Undecidability of the Halting Problem for PVS0). *If there is a surjective function from $\mathcal{V}al$ to expressions of PVS0, then there is no function "halt" that verifies whether an expression is terminating according to the predicate of semantic termination:*

$$\exists\ (\mathit{Val2Expr} : \mathcal{V}al \to \mathcal{E}xpr) : \forall(\mathit{expr} : \mathcal{E}xpr) :\ \exists(\nu : \mathcal{V}al) :\ \mathit{Val2Expr}(\nu) = \mathit{expr}$$
$$\Rightarrow \qquad \neg\exists\ (\mathit{halt} : \mathcal{E}xpr \to \mathtt{bool}) : \forall\ e_f : \mathit{halt}(e_f) \Leftrightarrow T_\varepsilon(e_f)$$

The existence of a surjective mapping from the values to the PVS0 expressions is a natural assumption that will hold whenever the non interpreted type $\mathcal{V}al$ is an infinite enumerable set. The formalization proceeds as explained below.

Proof. (formalization sketch) Suppose: $\exists\ (\mathit{Val2Expr} : \mathcal{V}al \to \mathcal{E}xpr) : \forall(\mathit{expr} : \mathcal{E}xpr) :\ \exists(\nu : \mathcal{V}al) : \mathit{to\_}\mathcal{E}\mathit{xpr}(\nu) = \mathit{expr}$. To obtain a contradition, suppose that:

$$\exists\ (\mathit{halt} : \mathcal{E}xpr \to \mathtt{bool}) : \forall\ e_f : \mathit{halt}(e_f) \Leftrightarrow T_\varepsilon(e_f)$$

Thus, there exists a function *halt* with such a property. The expression e_f will be chosen as $m := \mathit{ite}(H(vr), \mathit{rec}(vr), vr)$. Therefore, it results in the following assertion: $\mathit{halt}(m) \Leftrightarrow T_\varepsilon(m)$.

Now, both cases $\mathit{halt}(m)$ and $\neg\mathit{halt}(m)$ are analysed.

Case *halt(m).* If $\forall\ e_f : \mathit{halt}(e_f) \Leftrightarrow T_\varepsilon(e_f)$ holds for an expression e_f, it also holds for an expression $\mathit{ite}(H(vr), \mathit{rec}(vr), vr)$, where $H^\mathfrak{I} := \mathit{halt} \circ \mathit{Val2Expr}$ (the symbol $\circ$ denotes function composition). In this case, $m = \mathit{ite}(H(vr), \mathit{rec}(vr), vr)$.

It results in the assertive $(\mathit{halt}(m) \Rightarrow T_\varepsilon(m)) \wedge (T_\varepsilon(m) \Rightarrow \mathit{halt}(m))$. By the definition of T_ε, thus, $\forall\beta, \exists\nu : \varepsilon(m, m, \beta, \nu)$.

Choosing $\beta := M_0$, where $\mathit{Val2Expr}(M_0) = m$ (the value M_0 exists since *Val2Expr* is surjective), $\varepsilon(m, m, \beta, \nu)$ goes to an infinite process for each value ν and, therefore, the value ν does not exists, which is a contradiction.

However, the proof assistant PVS does not allow this kind of proof because of infinite process of ε. In this case, it was applied the equivalence $T_\varepsilon(m) \equiv T_\zeta(m)$. Therefore, $T_\zeta(m)$ was used:

$$\exists\ \mu,\ \forall\ (\beta,\ cc : \mathit{Valid\_cc}(m),\ \nu) :$$
$$\varepsilon(\mathit{get\_arg}(\mathit{rec\_expr}(cc)), m, \beta, \nu) \wedge \mathit{EC}(m, \mathit{cnds}(cc), \beta) \Rightarrow \mu(\nu) \prec \mu(\beta)$$

Thus, there exists μ such that :

$$\forall\ (\beta,\ cc : \mathit{Valid\_cc}(m),\ \nu) :$$
$$\varepsilon(\mathit{get\_arg}(\mathit{rec\_expr}(cc)), m, \beta, \nu) \wedge \mathit{EC}(m, \mathit{cnds}(cc), \beta) \Rightarrow \mu(\nu) \prec \mu(\beta)$$

Here, $\beta := M_0$ is chosen with, $cc := \langle rec\_expr := rec(vr), cdns := [H(vr)], path := [1]\rangle$ and $\nu := M_0$.

Simplifying the definitions ε and $EC : H^{\mathfrak{I}}(M_0) \Rightarrow \mu(M_0) \prec \mu(M_0)$. Therefore: $halt(m) \Rightarrow \mu(M_0) \prec \mu(M_0)$. Finally, by the assumption of $halt(m)$: $\mu(M_0) \prec \mu(M_0)$, that is a contradition, because the relation $\prec$ is well-founded.

Case $\neg halt(m)$. One has that $\neg T_\varepsilon(m)$ holds. By expanding the definition of T_ε: $\neg\forall\ \beta,\ \exists\ \nu \in \mathcal{V}al :\ \varepsilon(m, m, \beta, \nu)$, which is equivalent to: $\exists\ \beta,\ \forall\ \nu \in \mathcal{V}al :\ \neg\varepsilon(m, m, \beta, \nu)$. Thus, there exists β, such that: $\forall\ \nu \in \mathcal{V}al :\ \neg\varepsilon(m, m, \beta, \nu)$. Expanding the definition of ε and considering $\neg halt(m)$: $\forall\ \nu \in \mathcal{V}al :\ \neg\varepsilon(vr, m, \beta, \nu)$.

To conclude, a value ν different of β is chosen, which generates a contradiction. □

4 Formalization of the Turing completeness of PVS0

Since undecidability of the halting problem for restricted and non expressive computational models is irrelevant, it is also necessary to prove that PVS0 is a Turing-complete language. The reader could argue that Turing completeness implies undecidability of the halting problem, but our intention was to prove in an explicit manner that this theorem holds for the PVS0 language.

To formalize Turing completeness it is proved that PVS0 encodes the basic recursive functions: zero, successor and projections as well as addition, multiplication and the function χ (that is, $\chi(x, y) = 1$ if $x \leq y$ else 0), and in addition that it is closed under the operations of composition and minimization.

Lemma 2 (Turing Completeness of PVS0). *If there is a surjective function that encodes each* PVS0 *expression as a value in Val and, there is a codification of $\mathcal{V}al$ using natural numbers through a bijective function, then the language* PVS0 *computes all partial recursive functions:*

$$\begin{aligned} \exists\,(\mathit{Val2Expr} : \mathcal{V}al \to \mathcal{E}xpr) : \forall(\mathit{expr} : \mathcal{E}xpr) :\ \exists(\nu : \mathcal{V}al) :\ \mathit{Val2Expr}(\nu) = \mathit{expr} \\ \wedge\quad \exists\,(\mathit{Val2Nat} : \mathcal{V}al \to \mathbb{N}) :\ \mathit{Val2Nat\ is\ bijective} \\ \Rightarrow \qquad \texttt{PVS0}\ \mathit{is\ Turing\text{-}Complete} \end{aligned}$$

Proof. If there exist a bijective function $Val2Nat : \mathcal{V}al \to \mathbb{N}$, then it is possible to encode successor, addition and multiplication. Considering *Val2Nat* the bijection between $\mathcal{V}al$ and $\mathbb{N}$ and *Nat2Val* the inverse function. The symbols S, P and M implement successor, addition and multiplication respectively:

- $S^{\mathfrak{I}}(v) = Nat2Val(\,Val2Nat(v) + 1)$
- $P^{\mathfrak{I}}(v_1, v_2) = Nat2Val(\,Val2Nat(v_1) + Val2Nat(v_2))$
- $M^{\mathfrak{I}}(v_1, v_2) = Nat2Val(\,Val2Nat(v_1) \times Val2Nat(v_2))$

There is a bijective function between $\mathbb{N}$ and non empty lists of $\mathbb{N}$. Thus, there exists a bijective function between $\mathcal{V}al$ and non empty lists of $\mathbb{N}$. Let call this function *to_List* and consider $rem(a, b)$ as the remainder of a and b; $nth(l, i)$ as the i^{th} element of the list l (starting from 0 until the length l minus one)

in a list l; and *length* as the number of elements in a list, the projection P_i is implemented as:

$$P_i^{\mathfrak{I}}(a,b) = nth(to\_List(a), rem(length(to\_List(a)), Val2Nat(b)))$$

The function χ is built as:

$$C_{hi}^{\mathfrak{I}}(a,b) = if\ Val2Nat(a) \leq Val2Nat(b)\ then\ Nat2Val(1)\ else\ Nat2Val(0)$$

For the composition, consider $Val2Expr : \mathcal{V}al \to \mathcal{E}xpr$ as a surjective function and the function $Expr2Val : \mathcal{E}xpr \to \mathcal{V}al$ given as:

$$Expr2Val(e) = \texttt{CHOOSE}(\{v : \mathcal{V}al \,|\, Val2Expr(v) = e\})$$

where `CHOOSE` is a function that chooses a value from a non empty set. Consider also the PVS functions (`PVSO` operators):

- $O^{\mathfrak{I}}(a,b) = \texttt{CHOOSE}(\{v : \mathcal{V}al | \texttt{IF}\ \exists\ t : \varepsilon(Val2Expr(a), Val2Expr(a), b, t)\ \texttt{THEN}\ \varepsilon(Val2Expr(a), Val2Expr(a), b, v)\ \texttt{ELSE}\ v = Nat2Val(0)\})$
- $O_{aux}^{\mathfrak{I}}(a,b) = \texttt{CHOOSE}(\{v : \mathcal{V}al | \texttt{IF}\ \exists\ t : \varepsilon(Val2Expr(a), Val2Expr(a), b, t)\ \texttt{THEN}\ v = Nat2Val(1)\ \texttt{ELSE}\ v = Nat2Val(0)\})$

Thus, composition is built as:

$$\begin{aligned} Comp(f,g) := ite(&O_{aux}(Expr2Val(g), vr),\\ &ite(O_{aux}(Expr2Val(f), O(Expr2Val(g), vr)),\\ &\quad O(Expr2Val(f), O(Expr2Val(g), vr)), rec(vr)), rec(vr)) \end{aligned}$$

The branching instruction *ite* considers $Nat2Val(0)$ as false and any other different value as true.

The unary operator *min* (in $OP1$) is specified in PVS as the predicate $min : [\mathcal{E}xpr \times \mathcal{V}al \times \mathcal{V}al] \to Bool$, that specifies minimization for an expression e:

$$\begin{aligned} min(\langle e,t,v\rangle) \quad := \quad & \texttt{IF}\ \varepsilon(e,e,t,Nat2Val(0))\ \texttt{THEN}\ t = v\\ & \texttt{ELSE}\ min(\langle e, Nat2Val(\,Val2Nat(t)+1), v\rangle) \end{aligned}$$

In addition, using *min*, a PVS predicate M_{aux} and an operator M_{in} are specified for verifying the existence and verifying and giving the minimum:

- $M_{aux}^{\mathfrak{I}}(a,b) = \texttt{CHOOSE}(\{v : \mathcal{V}al | \texttt{IF}\ \exists\ t\ min(\langle\, Val2Expr(a), b, t\rangle)\ \texttt{THEN}\ v = Nat2Val(1)\ \texttt{ELSE}\ v = Nat2Val(0)\})$
- $M_{in}^{\mathfrak{I}}(a,b) = \texttt{CHOOSE}(\{v : \mathcal{V}al | \texttt{IF}\ \exists\ t\ min(\langle\, Val2Expr(a), b, t\rangle)\ \texttt{THEN}\ min(\langle\, Val2Expr(a), b, v\rangle)\ \texttt{ELSE}\ v = Nat2Val(0)\})$

Finally, minimization of `PVSO` functions is implemented in `PVSO` as:

$$MIN(f) := ite(M_{aux}(Expr2Val(f), vr), M_{in}(Expr2Val(f), vr), rec(vr))$$

To prove that MIN minimizes indeed, it is necessary to prove that the predicate *min* holds for values v such that the evaluation of the expression e with input v results in $Nat2Val(0)$: $\forall(e,t,v) : min(\langle e,t,v\rangle) \Rightarrow \varepsilon(e,e,v,Nat2Val(0))$.

One proceeds by induction on *min*'s structure. It means one considers that the property holds for the recursion to show that holds for the min procedure.

- Case $\varepsilon(e, e, t, Nat2Val(0))$ then $t = v$. Thus $\varepsilon(e, e, v, Nat2Val(0))$.
- Case $\neg\varepsilon(e, e, t, Nat2Val(0))$ then $min(\langle e, Nat2Val(\, Val2Nat(t) + 1), v\rangle)$. By hypothesis of induction, $\varepsilon(e, e, v, Nat2Val(0))$.

To formalize that the predicate min indeed minimizes, it is required to prove the following property, that states that min holds for a minimum value v:
$\forall(e, t, v) : min(\langle e, t, v\rangle) \Rightarrow \forall h : Val2Nat(t) \leq Val2Nat(h) < Val2Nat(v) \Rightarrow$
$\neg\varepsilon(e, e, h, Nat2Val(0))$
The proof is also by induction on min's structure.

- Case $\varepsilon(e, e, t, Nat2Val(0))$ then $t = v$. Thus $\forall h : Val2Nat(v) \leq Val2Nat(h) < Val2Nat(v) \Rightarrow \neg\varepsilon(e, e, h, Nat2Val(0))$. The assertion $Val2Nat(v) \leq Val2Nat(h) < Val2Nat(v)$ is false, so it is true that $\forall h : Val2Nat(v) \leq Val2Nat(h) < Val2Nat(v) \Rightarrow \neg\varepsilon(e, e, h, Nat2Val(0))$.
- Case $\neg\varepsilon(e, e, t, Nat2Val(0))$ then $min(\langle e, Nat2Val(\, Val2Nat(t) + 1), v\rangle)$. By hypothesis of induction,
  $\forall h : Val2Nat(t) + 1 \leq Val2Nat(h) < Val2Nat(v) \Rightarrow \neg\varepsilon(e, e, h, Nat2Val(0))$. If $t = h$, then $\neg\varepsilon(e, e, h, Nat2Val(0))$ holds. Then, $\forall h : Val2Nat(t) \leq Val2Nat(h) < to\_Nat(v) \Rightarrow \neg\varepsilon(e, e, h, Nat2Val(0))$. □

5 Conclusion and Future Work

A formalization of the undecidability of the Halting Problem for a minimal functional language `PVS0`, that has also been formally proved to be Turing complete, was presented. In the formalization of undecidability, the equivalence between termination TCCs (Turing's ranking function) and semantic termination notions was crucial to guarantee proof convergence (avoiding expanding the definition of ε indefinitely) when trying to find a value that does not exists in `PVS0`, and the program executes with itself as input. In this manner the required contradiction can be effectively obtained. Also, the assumption that there exists a surjective mapping from the non interpreted type `T` and `PVS0` expressions (programs) built over this type was important to guarantee the *Gödelization* of the `PVS0` programs. In addition, The cardinality of the non interpreted type `T` must be at least ω (the cardinality of naturals). For the proof of Turing completeness, the cardinality of `T` should be ω as well because this proof works with recursive functions and consequently with naturals.

Future work includes extending the restricted syntax of `PVS0`. Among other drawbacks, this language allows only specification of (recursive) functions with only one parameter over a non interpreted type `T`, being the output a value of the same type. This simplifies the specification and formalization of mechanisms for checking termination of programs and related properties such as equivalence between different technologies for automating termination, but in practice, such as in the PVS specification language, one uses more elaborated expressions in which operators give outputs of arbitrary types and deal with tuples of parameters of different types. Thus, the desired extension should allow additional flexibilities

such as manipulation of tuples of parameters of different types, other branching instructions such as CASE OF instructions, assigning instructions such as LET IN, etc. The extended language will facilitate the translation from real programming languages and consideration of other computational properties such as those related with complexity. Thus, the programs specified in the extended language as well as adaptations of the technologies for the automation of termination will be translated into the setting of the `PVS0` language in a conservative manner, and in this way, all properties proved for `PVS0` would be inherited for the extended language.

6 Acknowledgments

We would like to thank Cesar Muñoz and Mariano Moscato for meaningful contributions on this work. To both them for the development of the kernel of the language `PVS0` and to the former for pointing out the importance of formalizing Turing completeness of `PVS0`.

References

1. Andréia B. Avelar. *Formalização da automação da terminação através de grafos com matrizes de medida*. PhD thesis, Universidade de Brasília, Departamento de Matemática, Brasília, Distrito Federal, Brasil, 2015.
2. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving thread termination. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 320–330, 2007.
3. Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. Lexicographic Termination Proving. In *Proceedings Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, joint TACAS ETAPS*, volume 7795 of *Springer LNCS*, pages 47–61, 2013.
4. Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings Logic for Programming, Artificial Intelligence, and Reasoning LPAR*, volume 3452 of *Springer LNCS*, pages 301–331, 2004.
5. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of POPL 2001: 28th ACM SIGPLAN-SIGACT Symp. on Princ. of Programming Lang.*, pages 81–92, 2001.
6. Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *Proceedings Computer Aided Verification CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proc.*, volume 4144 of *Springer LNCS*, pages 401–414, 2006.
7. Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42(1):230–265, 1937.
8. Alan M. Turing. Checking a large routine. In Martin Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.