

Port-Graphs for Access Control Verification

Jorge Iglésias

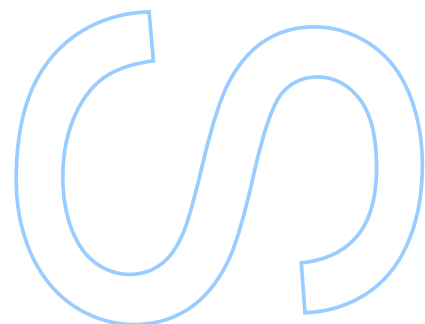
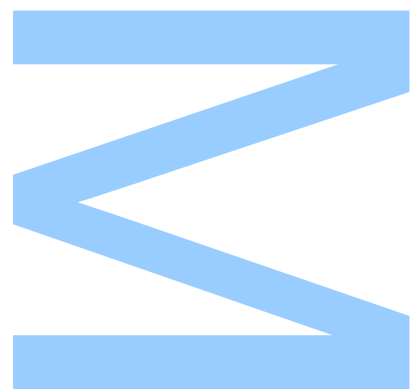
Mestrado em Ciência de Computadores

[Departamento de Ciência de Computadores](#)

2021

Orientador

[Prof. Dra. Sandra Alves](#), Faculdade de Ciências

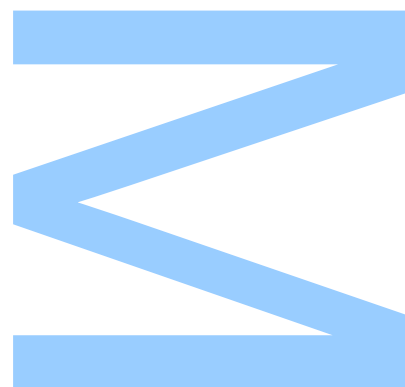




Todas as correções determinadas
pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ____/____/____



UNIVERSIDADE DO PORTO

MASTERS THESIS

Port-Graphs for Access Control Verification

Author:

Jorge IGLÉSIAS

Supervisor:

Sandra ALVES

*A thesis submitted in fulfilment of the requirements
for the degree of MSc. in Computer Science*

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

December 29, 2021

Acknowledgements

I want to thank my supervisor, professor Sandra Alves, for all the guidance, patience and understanding she showed me in every step of writing this thesis.

I want to thank Bruno Pinaud, from the University of Bordeaux, developer of PORGY, for always being available to help whenever I had troubles with PORGY.

I want to thank CRACS for the funding for this thesis through research scholarship AE2017-0191 (NanoStima-RL3 - CRACS) of project NanoSTIMA - Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics (NORTE-01-0145-FEDER-000016).

Last but not least, I want to thank all my family and friends, especially my mom, for all the love and support, without which I could not have written this thesis.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto

Departamento de Ciência de Computadores

MSc. in Computer Science

Port-Graphs for Access Control Verification

by [Jorge IGLÉSIAS](#)

We design and implement a graph-based framework for the visualisation and analysis of obligations in access control policies. We consider obligation policies in CBACO, the category-based access control model, which has been shown to subsume many of the most well known access control models such as MAC, DAC, RBAC. CBACO is an extension of the CBAC metamodel that deals with obligations. We describe the implementation of the proposed model in PORGY, a strategy driven graph-rewriting tool, based on the theory of port-graphs. CBACO policies allow for dynamic behavior in the modelled systems, which is implemented using the strategy language and rewrite rules of PORGY.

Key Words: Security Policies, Access Control, Obligations, Rewriting, Port Graphs

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto

Departamento de Ciência de Computadores

Mestrado em Ciência de Computadores

Grafos com Portas para Verificação de Controlo de Acessos

por [Jorge IGLÉSIAS](#)

Um modelo, baseado em grafos, para a visualização e análise de obrigações em políticas de controlo de acessos é introduzido e implementado. Consideramos políticas com obrigações no CBACO, o modelo de controlo de acessos baseado em categorias, que foi mostrado reproduzir muitos dos mais conhecidos modelos de controlo de acessos como MAC, DAC, RBAC. O CBACO é uma extensão do metamodelo CBAC que lida com obrigações. A implementação do modelo é descrita no PORGY, uma ferramenta de reescrita de grafos orientada por estratégias, baseada na teoria de grafos com portas. As políticas CBACO permitem comportamentos dinâmicos nos sistemas modelados, o que é implementado usando a linguagem de estratégias e regras de reescrita do PORGY.

Palavras-Chave: Políticas de Segurança, Controlo de Acessos, Obrigações, Reescrita, Grafos com Portas

Contents

Acknowledgements	iii
Abstract	v
Resumo	vii
Contents	ix
List of Figures	xi
List of Tables	xiii
1 Introduction	1
2 Background	3
2.1 Basic notions	3
2.2 Category-based access control policies	5
2.2.1 Permissions and prohibitions	5
2.2.2 Events	7
2.2.3 Obligations	9
2.3 CBAC representation using graphs	11
2.3.1 CBAC without prohibitions	11
2.3.2 CBAC with prohibitions	14
2.3.3 Dynamic CBAC policies	15
2.4 PORGY	16
2.4.1 Labeled Port Graphs	16
2.4.2 Labeled Port Graph Rewriting	19
2.4.3 Strategy Driven Rewriting	22
3 Graphs for dynamic CBACO policies	31
3.1 State of Duties	31
3.2 Static CBACO policies	32
3.3 Dynamic CBACO policies	38
3.3.1 Scenario-specific rewrite rules	38
3.3.2 General rewrite rules	38
3.3.2.1 Auxiliary rewrite rules	38
3.3.2.2 Obligation rewrite rules	39

3.3.2.3	Visualization and queries rewrite rules	39
4	Representing CBAC(O) in PORGY	41
4.1	Graphical CBAC in PORGY	41
4.1.1	Entities	41
4.1.2	Relations	42
4.2	Graphical CBACO in PORGY	43
4.2.1	Entities related to obligations	43
4.2.2	Relations related to obligations	44
4.2.3	Color coding	44
5	Dynamic CBAC(O) in PORGY	49
5.1	PCA	49
5.2	Auxiliary edges	53
5.2.1	Auxiliary nodes and edges for <i>UNDET</i>	59
5.3	Duties	65
5.4	Views and Queries	87
6	Related work	95
7	Conclusion	99
A	Importing policy graphs into PORGY	101
A.1	CSV import in Tulip	101
A.2	Getting the graph ready in PORGY	105
A.2.1	Visual rewrite rules and strategies	105
A.2.2	Python script	109
B	Value variables in PORGY	113
	Bibliography	115

List of Figures

2.1	Port graph	18
2.2	Port graph	20
2.3	Rewrite rule	21
2.4	Result of a rewriting step	23
2.5	Located port graph	23
2.6	Result of port graph rewriting using strategies	27
2.7	Result of port graph rewriting using strategies (first step)	29
2.8	Result of port graph rewriting using strategies (second step)	29
4.1	CBAC policy representation	43
4.2	CBACO policy representation	46
4.3	CBACO policy representations	46
5.1	Starting graph	50
5.2	Rewrite rule conRegisteredNurse	51
5.3	Rewrite rule rmPC	52
5.4	Result of applying pca to starting graph	53
5.5	Result of applying pca after changing qualification of C. Espinosa	53
5.6	Rewrite rule auxPCpyy	55
5.7	Rewrite rules	56
5.8	Rewrite rule rmauxPCpyy	58
5.9	CBAC policy representation	60
5.10	Rewrite rule auxPryy	60
5.11	Rewrite rule auxPPruyy	62
5.12	Rewrite rule rmauxPPruyy	63
5.13	Result of applying strategy auxPPru	65
5.14	Rewrite rule nextE	67
5.15	Rewrite rule closeH	67
5.16	Rewrite rule instDbotgggg	69
5.17	Rewrite rule instDgggg	71
5.18	Rewrite rule instauxDgggg	72
5.19	Rewrite rule updDpvgggg	75
5.20	Rewrite rule updauxDpvgggg	75
5.21	Rewrite rule updDpvbotgggg	76
5.22	Rewrite rule updauxDpvbotgggg	76
5.23	Rewrite rule updDpfgggg	79
5.24	Rewrite rule updDpfbotbotgggg	79
5.25	Rewrite rule updDpfbotgggg	80

5.26 Rewrite rule updDffgggg	83
5.27 Rewrite rule updauxDffgggg	84
5.28 Rewrite rule updDffbotgggg	85
5.29 Rewrite rule updauxDffbotgggg	85
5.30 Rewrite rule hidePy	88
5.31 Rewrite rule showPy	89
5.32 Rewrite rule hidePCyy	90
5.33 Rewrite rule showPCyy	90
5.34 CBACO policy representation	92
5.35 Result of applying strategy hideObl	92
5.36 Result of applying strategy hideAuth	92
5.37 CBAC policy representation	93
5.38 Result of applying strategy Undet	93
A.1 Example of CSV file with nodes	102
A.2 Example of CSV file with edges	102
A.3 Tulip CSV import window	103
A.4 Policy graph in Tulip	104
A.5 Porgy Tlp import	105
A.6 Graph in PORGY	105
A.7 Rewrite rule visualCPra	106
A.8 Rewrite rule visualPp	107
A.9 Graph after applying strategy visual	109
A.10 Tulip Python IDE in PORGY	111
A.11 Graph after using Porgy script	111
B.1 Algorithm tab in PORGY	113

List of Tables

2.1	Syntax of strategy language	24
4.1	Properties of nodes in a CBAC representation	42
4.2	Properties of new nodes in a CBACO representation	43

Chapter 1

Introduction

Access Control is the area of Computer Science that studies how to model and implement control of access to resources of a system, through policies that manage who (or what) can do what to which resource. The main goal is to ensure that every principal (person, process, etc.) has access to every resource that is needed for its tasks, while ensuring it does not have access to resources it should not have. With the ever growing complexity of systems this seemingly simple task becomes more difficult while at the same time more essential to ensure the integrity of systems. A number of frameworks have been proposed and used to model, analyse and implement such policies. Amongst them the most popular are Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC). RBAC is a model centered on the notion of roles, in which both principals and permissions are assigned to roles. Additionally there can be a hierarchy of roles (HRBAC). RBAC is static, in the sense that there is the need for an administrator to maintain the policy. As usual the existence of different frameworks means each of them has its strengths and weaknesses while ensuring that there is no single way of reasoning on all of them. Category Based Access Control (CBAC) [1] is a metamodel centered on the notion of category which has been shown to subsume many of the most used frameworks. A category is a class of entities that share some property. Classic types of groupings used in access control, like a role, a security clearance, a discrete measure of trust, etc., are particular instances of the more general notion of category. Categories can be defined on the basis of user attributes, geographical constraints, resource attributes, etc. As in RBAC both permissions and users are assigned to categories and there can be a hierarchy between categories. In contrast to RBAC, CBAC is a dynamic model, both principal- and permission-category assignment can be updated based on the state of the

system without the need for an administrator to intervene. CBAC deals with permissions and prohibitions (what can or cannot be done), but in access control there is also the need to deal with obligations (what must be done). CBACO [2] is an extension of the CBAC framework that also handles obligations, that is, tasks that principals of the system must do in certain situations. While a lot of work has been done on the analysis of CBAC policies mostly using term rewriting, work that takes advantage of visual formalism is more scarce. Having in mind that visual representations often have a positive impact on the understandability of concepts, especially for non-expert users, in this work we aim at extending an existing framework for the representation and analysis of CBAC policies using graphs [3] to CBACO and implement it. Given the dynamic nature of the CBACO model, the tool used to implement it has to be highly dynamic while having at the same time a strong theoretic background to assure correctness.

We choose to develop our graphical tool using PORGY [4], a visual modelling environment with specific simulation and verification tools that we believe to be well suited for the analysis of policies. PORGY is based on the theory of port-graphs [5] and implements a strategy driven language for port graph rewriting. In a port-graph, edges attach to nodes at specific connection points called ports. Rules and rewriting strategies are also viewed as port-graphs that can be rewritten into new rules, enabling the modelling of adaptive systems. Port-graphs have been used to model complex systems in the areas of biochemistry and interaction nets, as well as algorithm animation and games. The PORGY tool is built on top of Tulip [6], an information visualization framework dedicated to the analysis and visualization of relational data.

The work presented in this thesis has been partially published in [7].

The remainder of the thesis is organised as follows. In Chapter 2, we recall some basic notions, the CBAC metamodel and some of its extensions, the graph-based framework for the representation of CBAC policies and PORGY. Chapter 3 discusses a graphical model for the representation of CBACO and describes how rewrite rules can be used to implement the dynamics of the model. Chapter 4 shows how the representation of our graphical model is implemented in PORGY, and in Chapter 5 we show how the strategy language and rewrite rules of PORGY can be used to implement the dynamics of the modeled systems. In Chapter 6, we discuss related work, and in Chapter 7, conclusions are drawn and further work is suggested.

Chapter 2

Background

In the interest of self-containment we introduce, in this chapter, all the basic notions as well as the notions from previous works that led to the development of this one.

2.1 Basic notions

In this section we recall some basic notions that will be used throughout the thesis. We start with the notion of record.

Definition 2.1 (Record). Let a_i range over a set of attribute names and v_i range over a set of values, a record $r \in \mathcal{REC}$ is a term of the form $\{a_1 = v_1, \dots, a_n = v_n\}$ (which can also be seen as the set of pairs $\{(a_1, v_1), \dots, (a_n, v_n)\}$). The function Atts is defined such that $\text{Atts}(r) = \{a_i \mid (a_i, v_i) \in r\}$ and the notation $r.a_i$ is used to refer to value v_i .

Now we recall the notion of graphs, for more information we refer the reader to [8].

Definition 2.2 (Graphs and paths). A graph G is a tuple $\langle N_G, E_G \rangle$ where:

- N_G is a finite set of nodes (denoted $n_1, n_2, \dots$);
- E_G is a finite set of edges (denoted $e_1, e_2, \dots$) such that $E_G \subseteq \{\{n_i, n_j\} \mid n_i, n_j \in N_G \wedge n_i \neq n_j\}$.

Furthermore, a path of length d in G between nodes n_1 and n_{d+1} is a sequence of pairwise distinct nodes $n_1, n_2, \dots, n_{d+1}$ in N_G such that $\{n_i, n_{i+1}\} \in E_G$ for every $1 \leq i \leq d$.

This is the most basic definition for graphs, they can be extended to be directed, labelled, acyclic, etc. We will use some of these extensions in the following sections. Now we introduce the notions of relation and partial order.

Definition 2.3 (Relation). A n -ary relation R on sets $S_1, S_2, \dots, S_n$ is a subset of the Cartesian product $S_1 \times S_2 \times \dots \times S_n$. The fact that $(m_1, m_2, \dots, m_n) \in R$ can be otherwise conveyed using:

- prefix notation: $Rm_1m_2 \dots m_n$;
- postfix notation: $m_1m_2 \dots m_nR$;
- infix notation for binary relations: m_1Rm_2 .

When every S_i is the same set S we call it simply a n -ary relation on S .

Definition 2.4 (Partial order). Consider a set S . A partial order on S is a binary relation R that is:

- reflexive: for every $m \in S, mRm$;
- antisymmetric: for every $m, n \in S$, if mRn and nRm then $m = n$;
- transitive: for every $m, n, o \in S$, if mRn and nRo then mRo .

Finally, we recall some notions of term rewriting. These definitions follow closely the ones in [9], for more information we refer the reader to [10].

Definition 2.5 (Signatures, variables, terms, positions and substitutions). A signature $\mathcal{F}$ is a finite set of function symbols, each one with a fixed arity. The set $\mathcal{X}$ is the set of denumerable variables (denoted $X_1, X_2, \dots$). $T(\mathcal{F}, \mathcal{X})$, which represents the set of terms t that can be built from $\mathcal{F}$ and $\mathcal{X}$, can be identified by the set of finite trees where each node is labelled by a symbol in $\mathcal{F} \cup \mathcal{X}$ such that a node labelled $f \in \mathcal{F}$ has the same number of subtrees as the arity of f and variables are only found in leaves. Positions p are strings of integers that represent a path from the root of a tree to a node. $t|_p$ represents the subterm of t identified by the subtree of t at position p . Given a term u the notation $t[u]_p$ is used either to represent that u is a subterm of t or the substitution of $t|_p$ by u in t at position p . $\mathcal{V}(t)$ represents the set of variables in term t . If each variable only appears once in t that term is linear, if $\mathcal{V}(t) = \emptyset$ then t is ground. Variable substitutions are written as $\{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$, where each t_i is different from X_i , and are represented as Greek letters using postfix notation for their application.

Definition 2.6 (Rewrite rules). Given a signature $\mathcal{F}$, a term rewriting system is a set of rewrite rules $R = \{l_i \rightarrow r_i\}_{i \in I}$ such that $l_i, r_i \in T(\mathcal{F}, \mathcal{X})$, $l_i \notin \mathcal{X}$ and $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$. A

term t rewrites to term u at position p by rewrite rule $l \rightarrow r$ with substitution σ , written $t \xrightarrow[p]{l \rightarrow r} u$ or simply $t \rightarrow_R u$, if $t[l\sigma]_p$ and $u = t[r\sigma]_p$. Such term t is called reducible. A term that cannot be reduced by any rule in R is called irreducible and said to be in normal form.

We denote by $\rightarrow_R^*$ the transitive and reflexive closure of $\rightarrow_R$. Amongst the principal properties of a term rewriting system are confluence, termination and convergence. A term rewriting system R is confluent if for every pair of terms u, v for which exists term t such that $t \rightarrow_R^* u$ and $t \rightarrow_R^* v$ then there exists term s such that $u \rightarrow_R^* s$ and $v \rightarrow_R^* s$. A term rewriting system is terminating if there are no infinite rewriting sequences. A term rewriting system is convergent if it is both confluent and terminating.

2.2 Category-based access control policies

CBAC is a metamodel for access control policies based on the notion of category. A category is any way of grouping users of a system based on some property (age, function, expertise, etc.). Access control systems are often used to model permissions, prohibitions and obligations. We start by defining CBAC policies only with permissions and work our way to a more general model.

2.2.1 Permissions and prohibitions

A permission is a pair (a, r) where a is an action and r a resource. An authorisation is a triple (p, a, r) that associates principal p with a permission. In CBAC both principals and permissions are assigned to categories and from these assignments arise authorizations.

Definition 2.7 (CBAC with permissions). Given $\Gamma = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \subseteq \rangle$ where: $\mathcal{P}$ is a finite set of principals (denoted $p_1, p_2, \dots$), $\mathcal{C}$ is a finite set of categories (denoted $c_1, c_2, \dots$), $\mathcal{A}$ is a finite set of actions (denoted $a_1, a_2, \dots$), $\mathcal{R}$ is a finite set of resources (denoted $r_1, r_2, \dots$) and $\subseteq$ is a partial ordering between categories. A CBAC policy is a tuple $\langle \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR}, \Gamma \rangle$ where the relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{PAR}$ are defined as follows:

- $\mathcal{PCA} \subseteq \mathcal{P} \times \mathcal{C}$ is the principal-category assignment: $(p, c) \in \mathcal{PCA}$ iff p belongs to category c ;

- $\mathcal{ARCA} \subseteq \mathcal{A} \times \mathcal{R} \times \mathcal{C}$ is the permission-category assignment: $(a, r, c) \in \mathcal{ARCA}$ iff members of category c can perform action a on resource r ;
- $\mathcal{PAR} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{R}$ is the permission-principal assignment: $(p, a, r) \in \mathcal{PAR}$ iff principal p can perform action a on resource r ;

and the following axiom is satisfied.

$$\begin{aligned} \forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}, ((\exists c, c' \in \mathcal{C}, (p, c) \in \mathcal{PCA} \wedge c \subseteq c' \wedge (a, r, c') \in \mathcal{ARCA}) \\ \Leftrightarrow (p, a, r) \in \mathcal{PAR}) \end{aligned} \quad (2.1)$$

Definition 2.8 (CBAC with prohibitions). Given $\Gamma = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \subseteq \rangle$ a CBAC policy with prohibitions is a tuple $\langle \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR}, \mathcal{BARCA}, \mathcal{BAR}, \mathcal{UNDET}, \Gamma \rangle$ where the additional relations $\mathcal{BARCA}$, $\mathcal{BAR}$ and $\mathcal{UNDET}$ are defined as follows:

- $\mathcal{BARCA} \subseteq \mathcal{A} \times \mathcal{R} \times \mathcal{C}$ is the prohibition-category assignment: $(a, r, c) \in \mathcal{BARCA}$ iff members of category c cannot perform action a on resource r ;
- $\mathcal{BAR} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{R}$ is the prohibition-principal assignment: $(p, a, r) \in \mathcal{BAR}$ iff principal p cannot perform action a on resource r ;
- $\mathcal{UNDET} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{R}$ is the undetermined permission principal assignment: $(p, a, r) \in \mathcal{UNDET}$ iff principal p is neither authorized nor banned from performing action a on resource r ;

and, adding to the one described in Definition 2.7, we have the following axioms.

$$\begin{aligned} \forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}, ((\exists c, c' \in \mathcal{C}, (p, c) \in \mathcal{PCA} \wedge c' \subseteq c \wedge (a, r, c') \in \mathcal{BARCA}) \\ \Leftrightarrow (p, a, r) \in \mathcal{BAR}) \end{aligned} \quad (2.2)$$

$$\begin{aligned} \forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}, (((p, a, r) \notin \mathcal{PAR} \wedge (p, a, r) \notin \mathcal{BAR}) \\ \Leftrightarrow (p, a, r) \in \mathcal{UNDET}) \end{aligned} \quad (2.3)$$

$$\mathcal{PAR} \cap \mathcal{BAR} = \emptyset \quad (2.4)$$

Note that prohibitions are inherited through categories in the reverse order in which permissions are inherited.

We will now extend the CBAC models to deal with the notion of obligation. While authorizations are often static, obligations may depend on events of the system. For this reason we will start by defining events, event types, event histories and event intervals.

2.2.2 Events

The definitions in this section follow the notion on events defined in [11].

Definition 2.9 (Event Specifications). Consider $X \in \mathcal{X}$, $f \in \mathcal{F}$ and $l \in \mathcal{L}$ where: $\mathcal{X}$ is a set of term variables, $\mathcal{F}$ is a set of function symbols and $\mathcal{L}$ is a set of labels. Values and event specifications are defined as follows:

$$\begin{aligned} v &::= X \mid f(v_1, \dots, v_n) & n \geq 0 \\ \text{spec} &::= \{l_1 = v_1, \dots, l_n = v_n\} & n > 0 \end{aligned}$$

A constant a is a particular case of $f(v_1, \dots, v_n)$ where $n = 0$. The event specification $\{l_1 = v_1, \dots, l_n = v_n\}$ represents the structure with labels $l_1, \dots, l_n$ and values $v_1, \dots, v_n$, respectively. A value/event specification without term variables is called a ground value/event specification, denoted v° and spec° , respectively.

Definition 2.10 (Event). A (specific) event $e \in \mathcal{E}$ is a ground event specification spec° , which represents an actual action/happening that occurred in the system.

Definition 2.11 (Event scheme). An event scheme $ge \in \mathcal{G}$ represents a family of events and is given by an event specification spec , to which is associated a context $ge[X_1, \dots, X_n]$ where $X_1, \dots, X_n$ are the term variables in spec . Replacing $X_1, \dots, X_n$ with ground values $v_1^\circ, \dots, v_n^\circ$ will result in a ground event specification spec° .

Example 2.1. The act of doctor X_1 reading the record of patient X_2 can be represented by event scheme $\text{gen_read}[X_1, X_2]$.

$$\text{gen_read}[X_1, X_2] = \{\text{act} = \text{Read}, \text{subj} = X_1, \text{obj} = \text{Rec}(X_2)\}$$

By replacing X_1 with $P. \text{Cox}$ and X_2 with $J. \text{Lewis}$ we get the following specific event.

$$\{\text{act} = \text{Read}, \text{subj} = P. \text{Cox}, \text{obj} = \text{Rec}(J. \text{Lewis})\}$$

We now define an instantiation relation that allows us to associate specific events with event schemes.

Definition 2.12 (Instantiation). The relation $\vdash_\theta v^\varnothing :: v$ (resp. $\vdash_\theta \text{spec}^\varnothing :: \text{spec}$), where θ is a substitution, is defined as follows:

$$\frac{}{\vdash_{\{v^\varnothing/X\}} v^\varnothing :: X} \quad \frac{\vdash_{\theta_1} v_1^\varnothing :: v_1 \quad \cdots \quad \vdash_{\theta_n} v_n^\varnothing :: v_n}{\vdash_{\theta_1 \cup \dots \cup \theta_n} f(v_1^\varnothing, \dots, v_n^\varnothing) :: f(v_1, \dots, v_n)} \quad (n \geq 0)$$

$$\frac{\vdash_{\theta_1} v_1^\varnothing :: v_1 \quad \cdots \quad \vdash_{\theta_n} v_n^\varnothing :: v_n}{\vdash_{\theta_1 \cup \dots \cup \theta_n} \{l_1 = v_1^\varnothing, \dots, l_n = v_n^\varnothing, l_{n+1} = v_{n+1}^\varnothing, \dots, l_{n+k} = v_{n+k}^\varnothing\} :: \{l_1 = v_1, \dots, l_n = v_n\}}$$

In the rules above, when defining $\theta_1 \cup \dots \cup \theta_n$ we assume that $\theta_1, \dots, \theta_n$ are compatible, which means they do not assign different values to the same variable.

Definition 2.13 (Time expressions and constraints). Let c range over a set $\mathcal{S}$, partially ordered by $\leq$ and closed under $+$. The set of time expressions and time constraints, denoted $t \in \mathcal{T}$ and $tc \in \mathcal{TC}$ respectively, is defined as follows:

$$t ::= c \mid X \mid t + c \quad tc ::= t \mid t^+ \mid t^{+c}$$

Time constraints can be seen as intervals $[t_1, t_2]$, where:

- $t_2 = t_1$, if the time constraint is a time expression;
- $t_2 = \infty$, if the time constraint is of the form t_1^+ ;
- $t_2 = t_1 + c$, if the time constraint is of the form t_1^{+c} .

Definition 2.14 (Instantiation of time expressions and constraints). Consider a substitution $\sigma = \{c_1/X_1, \dots, c_n/X_n\}$. The instantiation $\llbracket \cdot \rrbracket \sigma$ of time expressions and time constraints is defined as follows:

$$\begin{aligned} \llbracket c \rrbracket \sigma &= c & \llbracket t + c \rrbracket \sigma &= \llbracket t \rrbracket \sigma + \llbracket c \rrbracket \sigma & \llbracket t^+ \rrbracket \sigma &= [\llbracket t \rrbracket \sigma, \infty] \\ \llbracket X \rrbracket \sigma &= \sigma(X) & \llbracket t^{+c} \rrbracket \sigma &= [\llbracket t \rrbracket \sigma, \llbracket t \rrbracket \sigma + \llbracket c \rrbracket \sigma] \end{aligned}$$

If $\mathcal{V}(t_1) \cup \mathcal{V}(t_2) \subseteq \text{dom}(\sigma)$ then $t_1 \preceq_\sigma t_2$ iff $\llbracket t_1 \rrbracket \sigma \leq \llbracket t_2 \rrbracket \sigma$. If t_1 and t_2 are ground we simply write $t_1 \preceq t_2$.

Definition 2.15 (Event history and event interval). An event history $h \in \mathcal{H}$ is a possible sequence of events of the system represented as a list of events of the form $[e_1, \dots, e_n]$ where every e_i is ground and has an attribute time with value t_i and such that if $i < j$ then $t_i \preceq t_j$. A subsequence of h is an event interval, denoted $I = (e_i, e_j, h)$, where e_i is the first event of the interval and e_j is the last one. We say that e_i opens the interval and e_j closes it. For every event e_k in h such that $t_i \preceq t_k \preceq t_j$ we say that $e_k \in (e_i, e_j, h)$.

Example 2.2. *The following is an example of what an event history could be.*

$$\begin{aligned}
 h = [& e_1 = \{\text{act} = \text{Read}, \text{subj} = \text{P. Cox}, \text{obj} = \text{Rec(J. Lewis)}, \text{time} = 0\}, \\
 & e_2 = \{\text{act} = \text{Code Blue}, \text{patient} = \text{J. Lewis}, \text{time} = 100\}, \\
 & e_3 = \{\text{act} = \text{Read}, \text{subj} = \text{C. Tuck}, \text{obj} = \text{Rec(J. Lewis)}, \text{time} = 120\}, \\
 & e_4 = \{\text{act} = \text{Perform}, \text{subj} = \text{C. Tuck}, \text{obj} = \text{Specimen collection}, \\
 & \quad \text{patient} = \text{J. Lewis}, \text{time} = 130\}, \\
 & e_5 = \{\text{act} = \text{Code Blue Stop}, \text{patient} = \text{J. Lewis}, \text{time} = 160\}, \\
 & e_6 = \{\text{act} = \text{Declare}, \text{subj} = \text{C. Tuck}, \text{obj} = \text{Admin-log}, \text{time} = 200\}]
 \end{aligned}$$

In this case the event interval (e_2, e_5) represents the subsequence of history where there was an active Code Blue referring to patient J. Lewis.

Having introduced the necessary event related notions we will now define the notions of obligation and duty and extend the CBAC metamodel to include them.

2.2.3 Obligations

The definitions in this subsection follow the notions defined in [2].

Definition 2.16 (Generic and Specific/Concrete Obligations). A generic obligation $o \in \mathcal{O}$ is an action $a \in \mathcal{A}$ on a resource $r \in \mathcal{R}$ that must be performed between two event schemes $ge_1, ge_2 \in \mathcal{G}$, represented as a tuple $o = (a, r, ge_1, ge_2)$. If there is no starting (resp. closing) event scheme, meaning the obligation can be performed at any time before ge_2 (resp. after ge_1), then ge_1 (resp. ge_2) is $\perp$. A specific/concrete obligation is an obligation where the event schemes ge_1, ge_2 are ground terms, therefore corresponding to specific events.

Like authorizations, obligations will be assigned to categories, but these might not be the same as authorization categories.

Definition 2.17 (Duty). A duty $d \in \mathcal{D}$, is a tuple (p, o) , where p is a principal, o is a concrete obligation (a, r, e_1, e_2) such that there exists a generic obligation (a, r, ge_1, ge_2) assigned to p and e_1, e_2 are instantiations of event schemes ge_1, ge_2 , respectively. If there is no starting (resp. closing) event then e_1 (resp. e_2) is $\perp$.

Duties can be in one of four states: *invalid* if, when the duty is issued, e_2 has already happened; *fulfilled* if the principal performed the action before e_2 ; *pending* if the principal did not perform the action but e_2 has not happened too; or *violated* if the action has not been performed by the principal and e_2 has already happened. These states are according

to some history. To facilitate the axiomatization of states of duties, that will be introduced later on, we assume that every history of events starts with an event $\perp$.

A CBACO is a CBAC policy with prohibitions and obligations. Because authorization and obligation categories might be different there is the need for a separate partial ordering on obligation categories, which we denote by $\subseteq_{\mathcal{O}}$.

Definition 2.18 (CBACO). Given $\Gamma = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \subseteq, \subseteq_{\mathcal{O}}, \mathcal{E}, \mathcal{G}, \mathcal{H} \rangle$ where, aside from what has already been defined in Definition 2.7: $\subseteq_{\mathcal{O}}$ is a partial ordering on obligation categories, $\mathcal{E}$ is a finite set of events (denoted $e_1, e_2, \dots$), $\mathcal{G}$ is a finite set of event schemes (denoted $ge_1, ge_2, \dots$) and $\mathcal{H}$ is a finite set of event histories (denoted $h_1, h_2, \dots$). The tuple $\langle \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR}, \mathcal{BARCA}, \mathcal{BAR}, \mathcal{UNDET}, \mathcal{OCA}, \mathcal{OPA}, \mathcal{DA}, \mathcal{ET}, \mathcal{EI}, \Gamma \rangle$ is a CBACO policy with prohibitions where the additional relations $\mathcal{OCA}$, $\mathcal{OPA}$, $\mathcal{DA}$, $\mathcal{ET}$ and $\mathcal{EI}$ are defined as follows:

- $\mathcal{OCA} \subseteq \mathcal{O} \times \mathcal{C}$ is the obligation-category assignment: $(o, c) \in \mathcal{OCA}$ iff there exists an obligation $o = (a, r, ge_1, ge_2)$ such that principals of category c are obliged to perform action a on resource r between events that are instantiations of event schemes ge_1 and ge_2 ;
- $\mathcal{OPA} \subseteq \mathcal{P} \times \mathcal{O}$ is the obligation-principal assignment: $(p, o) \in \mathcal{OPA}$ iff there exists an obligation $o = (a, r, ge_1, ge_2)$ such that principal p is obliged to perform action a on resource r between events that are instantiations of event schemes ge_1 and ge_2 ;
- $\mathcal{DA} \subseteq \mathcal{D}$ is the duty assignment: $d = (p, o) \in \mathcal{DA}$ iff there is a concrete obligation $o = (a, r, e_1, e_2)$ such that principal p is obliged to perform action a on resource r between events e_1 and e_2 ;
- $\mathcal{ET} \subseteq \mathcal{E} \times \mathcal{G}$ is the event instantiation relation: $(e, ge) \in \mathcal{ET}$ iff event e is an instance of event scheme ge , denoted $e :: ge$, according to an instantiation relation between events and event schemes, such as the one described in Definition 2.12;
- $\mathcal{EI} \subseteq \mathcal{E} \times \mathcal{E} \times \mathcal{H}$ is the event interval relation: $(e_1, e_2, h) \in \mathcal{EI}$ iff event e_2 closes the interval initiated by event e_1 in event history h .

And, adding to the ones described in Definitions 2.7 and 2.8, we have the following axioms.

$$\forall o \in \mathcal{O}((\exists c, c' \in \mathcal{C}, (p, c) \in \mathcal{PCA} \wedge c \subseteq_{\mathcal{O}} c' \wedge (o, c') \in \mathcal{OCA}) \Leftrightarrow (p, o) \in \mathcal{OPA}) \quad (2.5)$$

$$\begin{aligned} \forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}, \forall e_1, e_2 \in \mathcal{E}, ((\exists ge_1, ge_2 \in \mathcal{G}, (p, \overbrace{(a, r, ge_1, ge_2)}^o) \in \mathcal{OPA}, \\ e_1 :: ge_1, e_2 :: ge_2) \Leftrightarrow \underbrace{(p, (a, r, e_1, e_2))}_d \in \mathcal{DA}) \quad (2.6) \end{aligned}$$

2.3 CBAC representation using graphs

Having defined the CBAC metamodel in the previous section, now we will introduce a graph-based framework for its representation. This framework was first presented in [12] and dealt with CBAC policies without prohibitions, in [3] it was extended to include prohibitions. We start by introducing the framework that deals with CBAC policies without prohibitions.

2.3.1 CBAC without prohibitions

First we introduce the notions of policy graph and path.

Definition 2.19 (Policy graph). A policy graph G is a tuple $\langle N_G, E_G, ln, le \rangle$ with:

- N_G a finite set of nodes (denoted $n_1, n_2, \dots$);
- E_G a finite set of undirected edges (denoted $e_1, e_2, \dots$);
- $ln : N_G \rightarrow \mathcal{REC}$ a labeling function for nodes such that for every node $n \in N_G$, $ln(n).ent \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{A} \cup \mathcal{R}$ and $ln(n).type \in \{P, C, A_{\mathcal{R}}, R\}$ where:
 - $ln(n).type = P$ iff $ln(n).ent = p \in \mathcal{P}$;
 - $ln(n).type = C$ iff $ln(n).ent = c \in \mathcal{C}$;
 - $ln(n).type = A_{\mathcal{R}}$ iff $ln(n).ent = a \in \mathcal{A}$ with $\mathcal{R} = \{r_1, \dots, r_n\}$, the set of records to which the node corresponds, used only when it is ambiguous;
 - $ln(n).type = R$ iff $ln(n).ent = r \in \mathcal{R}$
- $le : E_G \rightarrow \mathcal{REC}$ a labeling function for edges such that for every edge $e \in E_G$, $le(e).adj = \{n_1, n_2\}$ gives the two nodes connected by the edge. We also define the function $type$ such that $type(e) = \{ln(n_1).type, ln(n_2).type\}$.

To simplify notation $type(e) = \{T_1, T_2\}$ will be written $T_1 T_2$. As edges are undirected there is no difference between types $T_1 T_2$ and $T_2 T_1$.

Definition 2.20 (Path). Given a policy graph $G = \langle N_G, E_G, ln, le \rangle$ a path of length d between nodes n_0 and n_d is a sequence of pairwise distinct nodes $n_0, n_1, \dots, n_d$ such that for every $1 \leq i \leq d$, $le(e).adj = \{n_{i-1}, n_i\}$ for some $e \in E_G$.

Note that edges are undirected, but if there is a $\subseteq$ relation then we might have that $c_1 \subseteq c_2$ but not $c_2 \subseteq c_1$ for some $c_1, c_2 \in \mathcal{C}$. For this reason edges of type CC have in their records a field *target* such that $n_1 \in le(e).target$ iff $le(e).adj = \{n_1, n_2\}$ and $ln(n_2).ent \subseteq ln(n_1).ent$. If categories are equivalent then $le(e).target = le(e).adj = \{n_1, n_2\}$.

Definition 2.21 (Constrained path). A constrained path of length d is a sequence of pairwise distinct nodes and edges, $n_0, e_1, n_1, \dots, e_d, n_d$, such that $le(e_i).adj = \{n_{i-1}, n_i\}$ and $n_i \in le(e_i).target$ when $le(e_i).target$ exists for every $1 \leq i \leq d$.

Definition 2.22 (Types of paths). Consider a path $n_0, n_1, \dots, n_d$ of length d such that $ln(n_i).type = T_i$ for every $0 \leq i \leq d$. The type of this path is given by the types of its edges $T_0 T_1, T_1 T_2, \dots, T_{d-1} T_d$.

The notation $type(n_0, n_1, \dots, n_d) = T_0 T_1, T_1 T_2, \dots, T_{d-1} T_d$ is used to denote that there is a path $n_0, n_1, \dots, n_d$ and its edges are of type $T_0 T_1, T_1 T_2, \dots, T_{d-1} T_d$.

If an edge e_i of the path is connecting two nodes n_{i-1} and n_i both of type C then its type will be $\overrightarrow{CC}$ if $n_i \in le(e_i).target$ and $\overleftarrow{CC}$ if $n_{i-1} \in le(e_i).target$. Note that an edge might be both of type $\overrightarrow{CC}$ and $\overleftarrow{CC}$.

Definition 2.23 (Redundant edges). Consider a policy graph $G = \langle N_G, E_G, ln, le \rangle$. An edge $e \in E_G$ is redundant if:

- $type(e) = PC$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = p$ and $ln(n_2).ent = c$ with $p \in \mathcal{P}, c \in \mathcal{C}$, and there is a path of type $PC, (\overrightarrow{CC})^*$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = CC$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c_1$ and $ln(n_2).ent = c_2$ with $c_1, c_2 \in \mathcal{C}$, and there is a path of type $(\overrightarrow{CC})^*$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = CA$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c$ and $ln(n_2).ent = a$ with $c \in \mathcal{C}, a \in \mathcal{A}$, and there is a path of type $(\overrightarrow{CC})^*, CA$ and length greater or equal to 2 connecting n_1 and n_2 .

The notion of redundant edges encodes the fact that there is a partial relation between categories that deems some edges unnecessary.

Definition 2.24 (Well-formed policy graph). A policy graph $G = \langle N_G, E_G, ln, le \rangle$ is well-formed iff for every $n_1, n_2 \in N_G$ if $ln(n_1).ent = ln(n_2).ent$ and $ln(n_1).type = ln(n_2).type$ then $n_1 = n_2$, for every $e_1, e_2 \in E_G$ if $le(e_1).adj = le(e_2).adj$ then $e_1 = e_2$, for every $e \in E_G$ with $le(e).adj = \{n_1, n_2\}$ one of the following is true:

- $ln(n_1).type = P \wedge ln(n_2).type = C$;
- $ln(n_1).type = C \wedge ln(n_2).type = C \wedge le(e).target \subseteq \{n_1, n_2\}$;
- $ln(n_1).type = C \wedge ln(n_2).type = A_R$;
- $ln(n_1).type = A_{\{r_1, \dots, r_n\}} \wedge ln(n_2).type = R \wedge ln(n_2).ent = r_i$;

and there are no redundant edges.

Definition 2.25 ($CBAC_G$). Given a well-formed policy graph $G = \langle N_G, E_G, ln, le \rangle$ the tuple $\langle \Gamma_G, PCA_G, ARCA_G, PAR_G \rangle$, denoted $CBAC_G$, where $\Gamma_G = \langle \mathcal{P}_G, \mathcal{C}_G, \mathcal{A}_G, \mathcal{R}_G, \subseteq \rangle$ and:

- $\mathcal{P}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = P\}$;
- $\mathcal{C}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = C\}$;
- $\mathcal{A}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = A\}$;
- $\mathcal{R}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = R\}$;
- $\subseteq = \{(ln(n_1).ent, ln(n_i).ent) \mid \exists n_2, \dots, n_i, type(n_1, n_2, \dots, n_i) = (\overrightarrow{CC})^*\}$;
- $PCA_G = \{(ln(n_1).ent, ln(n_2).ent) \mid type(n_1, n_2) = PC\}$;
- $ARCA_G = \{(ln(n_2).ent, ln(n_3).ent, ln(n_1).ent) \mid type(n_1, n_2, n_3) = CA, AR\}$;
- $PAR_G = \{(ln(n_1).ent, ln(n_3).ent, ln(n_4).ent) \mid \exists n_{21}, \dots, n_{2i}, type(n_1, n_{21}, \dots, n_{2i}, n_3, n_4) = PC, (\overrightarrow{CC})^*, CA, AR\}$;

defines the CBAC policy described by graph G .

Next we introduce the extended framework that deals with prohibitions.

2.3.2 CBAC with prohibitions

Because prohibitions are inherited by categories in the inverse direction relative to permissions we introduce the notion of constrained inverse path.

Definition 2.26 (Constrained inverse path). A constrained inverse path of length d is a sequence $n_0, e_1, n_1, \dots, e_d, n_d$, with pairwise distinct nodes and edges, such that for every $1 \leq i \leq d$, $le(e_i).adj = \{n_{i-1}, n_i\}$ and $n_{i-1} \in le(e_i).target$ when $le(e_i).target$ exists.

Edges of type CA will be used to represent both permissions and prohibitions. To distinguish them, an edge e representing a permission (resp. prohibition) will have attribute $auth$ with value A (resp. B) and we will use the notation $type(e) = CA^A$ (resp. CA^B). Because of this we need to update the definition of redundant edges of this type.

Definition 2.27 (Redundant CA edges). Given a policy graph with prohibitions $G = \langle N_G, E_G, ln, le \rangle$ an edge $e \in E_G$ such that $type(e) \in \{CA^A, CA^B\}$ is redundant if:

- $type(e) = CA^A$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c$ and $ln(n_2).ent = a$ with $c \in \mathcal{C}, a \in \mathcal{A}$ and $le(e).auth = A$, and there is a path of type $(\overrightarrow{CC})^*, CA^A$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = CA^B$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c$ and $ln(n_2).ent = a$ with $c \in \mathcal{C}, a \in \mathcal{A}$ and $le(e).auth = B$, and there is a path of type $(\overleftarrow{CC})^*, CA^B$ and length greater or equal to 2 connecting n_1 and n_2 .

Definition 2.28 (Well-formed policy graph with prohibitions). A policy graph with prohibitions $G = \langle N_G, E_G, ln, le \rangle$ is well-formed iff for every $n_1, n_2 \in N_G$ if $ln(n_1).ent = ln(n_2).ent$ and $ln(n_1).type = ln(n_2).type$ then $n_1 = n_2$, for every $e_1, e_2 \in E_G$ if $le(e_1).adj = le(e_2).adj$ then either $e_1 = e_2$ or e_1, e_2 have type CA and $le(e_1).auth \neq le(e_2).auth$, for every $e \in E_G$ with $le(e).adj = \{n_1, n_2\}$ one of the following is true:

- $ln(n_1).type = P \wedge ln(n_2).type = C$;
- $ln(n_1).type = C \wedge ln(n_2).type = C \wedge le(e).target \subseteq \{n_1, n_2\}$;
- $ln(n_1).type = C \wedge ln(n_2).type = A_{\mathcal{R}} \wedge le(e).auth \in \{A, B\}$;
- $ln(n_1).type = A_{\{r_1, \dots, r_n\}} \wedge ln(n_2).type = R \wedge ln(n_2).ent = r_i$;

and there are no redundant edges. Moreover if a constrained path and an inverse constrained path begin in the same node with type P and end in nodes of type A such that

the last edges of the paths are of type CA^A and CA^B , respectively, then the end node must be different. This means that a principal cannot be both able and unable to perform an action on a resource.

Definition 2.29 (CBAC_G with prohibitions). Given a well-formed policy graph with prohibitions G , the tuple $\langle \Gamma_G, \mathcal{PCA}_G, \mathcal{ARCA}_G, \mathcal{PAR}_G, \mathcal{BARCA}_G, \mathcal{BAR}_G, \mathcal{UNDET}_G \rangle$, denoted CBAC_G, where $\Gamma_G = \langle \mathcal{P}_G, \mathcal{C}_G, \mathcal{A}_G, \mathcal{R}_G, \subseteq \rangle$ and:

- $\mathcal{P}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = P\};$
- $\mathcal{C}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = C\};$
- $\mathcal{A}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = A\};$
- $\mathcal{R}_G = \{ln(n).ent \mid n \in N_G \wedge ln(n).type = R\};$
- $\subseteq = \{(ln(n_1).ent, ln(n_i).ent) \mid \exists n_2, \dots, n_i, type(n_1, n_2, \dots, n_i) = (\overrightarrow{CC})^*\};$
- $\mathcal{PCA}_G = \{(ln(n_1).ent, ln(n_2).ent) \mid type(n_1, n_2) = PC\};$
- $\mathcal{ARCA}_G = \{(ln(n_2).ent, ln(n_3).ent, ln(n_1).ent) \mid type(n_1, n_2, n_3) = CA^A, AR\};$
- $\mathcal{PAR}_G = \{(ln(n_1).ent, ln(n_3).ent, ln(n_4).ent) \mid \exists n_{21}, \dots, n_{2i},$
 $type(n_1, n_{21}, \dots, n_{2i}, n_3, n_4) = PC, (\overrightarrow{CC})^*, CA^A, AR\};$
- $\mathcal{BARCA}_G = \{(ln(n_2).ent, ln(n_3).ent, ln(n_1).ent) \mid type(n_1, n_2, n_3) = CA^B, AR\};$
- $\mathcal{BAR}_G = \{(ln(n_1).ent, ln(n_3).ent, ln(n_4).ent) \mid \exists n_{21}, \dots, n_{2i},$
 $type(n_1, n_{21}, \dots, n_{2i}, n_3, n_4) = PC, (\overleftarrow{CC})^*, CA^B, AR\};$
- $\mathcal{UNDET}_G = \{(ln(n_1).ent, ln(n_2).ent, ln(n_3).ent) \mid ln(n_1).type = P, ln(n_2).type =$
 $A, ln(n_3).type = R\} \setminus (\mathcal{PAR}_G \cup \mathcal{BAR}_G);$

defines the CBAC policy described by graph G .

2.3.3 Dynamic CBAC policies

Note that the framework presented so far comprises only a static representation of a CBAC policy. Now we introduce an extension of the framework to represent dynamic policies.

Definition 2.30 (Dynamic policy graph). A dynamic policy graph is a well-formed policy graph with prohibitions $G = \langle N_G, E_G, ln, le \rangle$ where every node $n \in N_G$ has in its record attribute *fun* with $ln(n).fun = R$ for some convergent rewrite system R such that:

- if $ln(n).type = P$ then $ln(n).fun$ defines a function *pca* (written $pca_{ln(n).fun}$) that returns the list of categories to which principal $ln(n).ent$ belongs to;
- if $ln(n).type = C$ then $ln(n).fun$ defines functions *arca* and *barca* (written $arca_{ln(n).fun}$ and $barca_{ln(n).fun}$, respectively) that return the list of pairs action/resource that are the permissions and prohibitions, respectively, assigned to category $ln(n).ent$.

Definition 2.31 (Dynamic policy graph correctness). A dynamic policy graph is correct with respect to a system state *st* iff:

- for every node $n \in N_G$ such that $ln(n).type = P$, $pca_{ln(n).fun} = \{c_1, \dots, c_m\}$ iff there exists an edge $e_i \in E_G$ for every $1 \leq i \leq m$ such that $le(e_i).adj = \{n, n_i\}$ and $ln(n_i).ent = c_i$;
- for every node $n \in N_G$ such that $ln(n).type = C$, $arca_{ln(n).fun} = \{(a_i, r_i) \mid 0 \leq i \leq m\}$ (resp. $barca_{ln(n).fun} = \{(a_i, r_i) \mid 0 \leq i \leq m\}$) iff there exist two edges $e_{i1}, e_{i2} \in E_G$ for every $0 \leq i \leq m$ such that $le(e_{i1}).auth = A$ (resp. $le(e_{i1}).auth = B$), $le(e_{i1}).adj = \{n, n_{i1}\}$, $ln(n_{i1}).ent = a_i$, $le(e_{i2}).adj = \{n_{i1}, n_{i2}\}$ and $ln(n_{i2}).ent = r_i$.

2.4 PORGY

Having introduced a graph-based framework to represent CBAC policies in the previous section, we will now introduce a tool for the visualization and rewriting of graphs, specifically port graphs. PORGY [4] uses port graph rewrite rules and strategies to control the evolution of port graphs, using this feature we will be able to encode all the dynamic properties of the CBACO metamodel. The notions on this section follow the strategic port graph rewriting presented in [4, 13]. For more information on PORGY we refer the reader to [14].

2.4.1 Labeled Port Graphs

Port graphs [5, 15] are a special kind of graph where edges connect to specific sites of the nodes called ports. Ports add expressiveness to graphs, for example ports can have

associated with them specific edges or nodes can have different types depending on the type of ports they have. We define port graphs using the notion of record over a specific signature, which is an extension of the basic notion of record in Definition 2.1.

Definition 2.32 (Signature). A signature Δ is a tuple $\langle \Delta^A, \mathcal{X}^A, \Delta^V, \mathcal{X}^V \rangle$ where:

- Δ^A a set of attributes;
- $\mathcal{X}^A$ a set of attribute variables;
- Δ^V a set of values;
- $\mathcal{X}^V$ a set of value variables.

Definition 2.33 (Record). A record over a signature $\Delta = \langle \Delta^A, \mathcal{X}^A, \Delta^V, \mathcal{X}^V \rangle$ is a set of pairs $r = \{(a_1, v_1), \dots, (a_n, v_n)\}$ with $n \geq 1$ where $a_i \in \Delta^A \cup \mathcal{X}^A, v_i \in \Delta^V \cup \mathcal{X}^V$ for all $1 \leq i \leq n$. Attributes are of atomic type, however it is easy to extend this definition so that instead of only atomic types one can use regular expressions as values of attributes. It is assumed that Δ^A contains attributes *Name*, *Interface*, *Arity*, *Attach* and *Connect*. Attribute *Name* identifies records in the sense that if $r_1.Name = r_2.Name$, then $Atts(r_1) = Atts(r_2)$. In other words, if two records have the same value for attribute *Name* then they have the same set of defined attributes, although their values can differ.

Definition 2.34 (Port graph). A port graph over a signature Δ is defined as a tuple $G = \langle N_G, P_G, E_G, \mathcal{L}_G \rangle$ with:

- N_G is a finite set of nodes (denoted $n_1, n_2, \dots$);
- P_G is a finite set of ports (denoted $p_1, p_2, \dots$);
- E_G is a finite set of undirected edges (denoted $e_1, e_2, \dots$);
- $\mathcal{L}_G$ is the labeling function that for every element of the sets described above returns the record associated with it such that:
 - for every node $n \in N_G$ the record contains attribute *Interface* and $\mathcal{L}_G(n).Interface$ returns the set of ports of n ;
 - for every port $p \in P_G$ the record contains attributes *Attach* and *Arity*, such that $\mathcal{L}_G(p).Arity$ gives the number of edges connected to that port and $\mathcal{L}_G(p).Attach$ returns the node n to which p belongs;

- for every edge $e \in E_G$ the record contains attribute *Connect* and $\mathcal{L}_G(e).Connect$ returns $\{p_1, p_2\}$ the ports that e is connected to.

Example 2.3. In Figure 2.1 we show an example of a port graph G where $N_G = \{n_1, n_2, n_3\}$, $P_G = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, $E_G = \{e_1, e_2\}$ and:

- $\mathcal{L}_G(n_1) = \{(Interface, \{p_1\}), (ent, J. Dorian), (type, P), (viewLabel, J. Dorian)\};$
- $\mathcal{L}_G(n_2) = \{(Interface, \{p_2, p_3, p_4\}), (ent, Dr(J. Lewis)), (type, C), (viewLabel, Dr(J. Lewis))\};$
- $\mathcal{L}_G(n_3) = \{(Interface, \{p_5, p_6, p_7\}), (ent, Doctor), (type, C), (viewLabel, Doctor)\};$
- $\mathcal{L}_G(p_1) = \{(Attach, n_1), (Arity, 1)\};$
- $\mathcal{L}_G(p_2) = \{(Attach, n_2), (Arity, 1)\};$
- $\mathcal{L}_G(p_3) = \{(Attach, n_2), (Arity, 0), (viewLabel, In)\};$
- $\mathcal{L}_G(p_4) = \{(Attach, n_2), (Arity, 1), (viewLabel, Out)\};$
- $\mathcal{L}_G(p_5) = \{(Attach, n_3), (Arity, 0)\};$
- $\mathcal{L}_G(p_6) = \{(Attach, n_3), (Arity, 1), (viewLabel, In)\};$
- $\mathcal{L}_G(p_7) = \{(Attach, n_3), (Arity, 0), (viewLabel, Out)\};$
- $\mathcal{L}_G(e_1) = \{(Connect, \{p_1, p_2\}), (type, PC)\};$
- $\mathcal{L}_G(e_2) = \{(Connect, \{p_4, p_6\}), (type, CC)\}.$

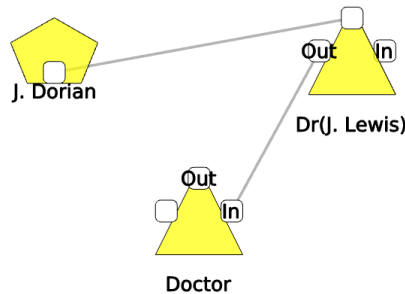


FIGURE 2.1: Port graph

2.4.2 Labeled Port Graph Rewriting

It is possible to change a port graph using rewrite rules. We will now introduce the notions related to port graph rewriting.

Definition 2.35 (Port graph morphism). Consider two port graphs $G = \langle N_G, P_G, E_G, \mathcal{L}_G \rangle$ and $H = \langle N_H, P_H, E_H, \mathcal{L}_H \rangle$ over the same signature Δ . A port graph morphism f is a set of injective functions $\langle f_N : N_G \rightarrow N_H, f_P : P_G \rightarrow P_H, f_E : E_G \rightarrow E_H \rangle$ together with two instantiation functions $f_1 : \mathcal{X}^A \rightarrow \Delta^A$ and $f_2 : \mathcal{X}^V \rightarrow \Delta^V$ that maps a port graph G into H such that:

- for every node n in N_G we have that $f_1(f_2(\mathcal{L}_G(n))) = \mathcal{L}_H(f_N(n))$;
- for every port p in P_G we have that $f_N(\mathcal{L}_G(p).Attach) = \mathcal{L}_H(f_P(p)).Attach$ and $f_1(f_2(\mathcal{L}_G(p))) = \mathcal{L}_H(f_P(p))$;
- for every edge e in E_G we have that $f_1(f_2(\mathcal{L}_G(e))) = \mathcal{L}_H(f_E(e))$ and, being $\bar{f}_P$ the extension of f_P to sets of ports, also that $\bar{f}_P(\mathcal{L}_G(e).Connect) = \mathcal{L}_H(f_E(e)).Connect$.

If there is such morphism then G and H are said isomorphic.

Example 2.4. The port graph H in Figure 2.2 is defined by $N_H = \{n_1, n_2, n_3\}$, $P_H = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, $E_H = \{e_1, e_2\}$ and:

- $\mathcal{L}_H(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2)\}$;
- $\mathcal{L}_H(n_2) = \{(Interface, \{p_2, p_3, p_4\}), (ent, v_3), (type, C), (viewLabel, v_4)\}$;
- $\mathcal{L}_H(n_3) = \{(Interface, \{p_5, p_6, p_7\}), (ent, v_5), (type, C), (viewLabel, v_6)\}$;
- $\mathcal{L}_H(p_1) = \{(Attach, n_1), (Arity, 1)\}$;
- $\mathcal{L}_H(p_2) = \{(Attach, n_2), (Arity, 1)\}$;
- $\mathcal{L}_H(p_3) = \{(Attach, n_2), (Arity, 0), (viewLabel, In)\}$;
- $\mathcal{L}_H(p_4) = \{(Attach, n_2), (Arity, 1), (viewLabel, Out)\}$;
- $\mathcal{L}_H(p_5) = \{(Attach, n_3), (Arity, 0)\}$;
- $\mathcal{L}_H(p_6) = \{(Attach, n_3), (Arity, 1), (viewLabel, In)\}$;
- $\mathcal{L}_H(p_7) = \{(Attach, n_3), (Arity, 0), (viewLabel, Out)\}$;

- $\mathcal{L}_H(e_1) = \{(Connect, \{p_1, p_2\}), (type, PC)\};$
- $\mathcal{L}_H(e_2) = \{(Connect, \{p_4, p_6\}), (type, CC)\}.$

This and the port graph G shown in Figure 2.1 are isomorphic with the port graph morphism f that maps H into G being defined as:

- $f_N(n_i) = n_i, i = 1, 2, 3;$
- $f_P(p_i) = p_i, i = 1, \dots, 7;$
- $f_E(e_i) = e_i, i = 1, 2;$
- $f_1 = id;$
- $f_2(v_i) = \begin{cases} J. Dorian & i = 1, 2 \\ Dr(J. Lewis) & i = 3, 4 \\ Doctor & i = 5, 6 \end{cases}.$

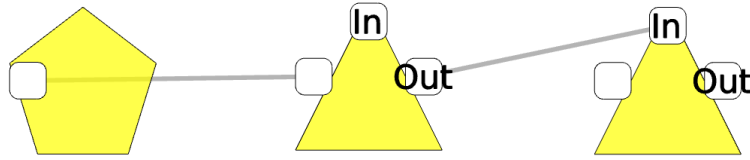


FIGURE 2.2: Port graph

Definition 2.36 (Rewrite rule). A rewrite rule is a port graph composed of two subgraphs, L and R , and a special node (called arrow node) that connects the two. It defines a way of rewriting graphs in the sense that, if there is a instance of L , it can be replaced by R and the arrow node defines the way to reconnect R to the rest of the graph. Ports in the arrow node have attribute *Type* that can have values bridge, blackhole or wire. These values define the rewriting in the following way:

- a port with *Type* bridge has exactly one edge connecting it to a port p of L and one or more connecting it to ports p_i of R . This type of port defines that edges of the

graph being rewritten that are connected to the image of p and are not in L should be reconnected to every p_i ;

- there is at most one port with *Type* blackhole and it only has edges connected to ports of L . It defines that edges connected to such ports should be erased;
- a port with *Type* wire has exactly two edges connecting it to two ports p_1, p_2 of L and it defines that there should be created an edge between every port p_{1i} that is connected to the image of p_1 and every port p_{2i} that is connected to the image of p_2 .

A rewrite rule $L \Rightarrow R$ matches a graph G if there is a morphism f such that $f(L) \subseteq G$ and the image of every port p in L not connected to the arrow node is not connected to any port of $G \setminus f(L)$. This last condition ensures that rewriting does not leave dangling edges.

Example 2.5. In Figure 2.3 it is shown an example of a rewrite rule where L is equal to the port graph in Figure 2.2, R is the same with the addition of an edge e_3 such that $\mathcal{L}_R(e_3) = \{(\text{Connect}, \{p_1, p_5\}), (\text{type}, \text{PC})\}$ and every port of the arrow node is of *Type* bridge.

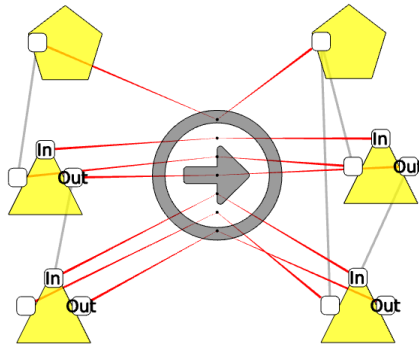


FIGURE 2.3: Rewrite rule

Definition 2.37 (Rewriting step). Consider a port graph G , a rewrite rule $L \Rightarrow R$ and a morphism f such that $f(L) \subseteq G$. A rewriting step from G by rule $L \Rightarrow R$ and morphism f , denoted $G \xrightarrow{f}_{L \Rightarrow R} G'$, is defined as follows:

1. the *build* phase: graph $f(R)$ is added to G ;
2. the *rewiring* phase: edges connecting $G \setminus f(L)$ to $f(L)$ are connected to $f(R)$ following the rules set by the *arrow node*;
3. the *deletion* phase: $f(L)$ is erased obtaining G' .

Example 2.6. The port graph G' in Figure 2.4 is the result of applying the rewrite rule shown in Figure 2.3 to the port graph G shown in Figure 2.1.

As stated in Example 2.5, L is equal to H and there is morphism f , described in Example 2.4, such that $f(H) \subseteq G$, hence $f(L) \subseteq G$. With this in mind, the rewriting is carried as follows:

1. the build phase: $f(R)$ is added to G where $f(R)$ is defined by $N_{f(R)} = \{n_1, n_2, n_3\}$, $P_{f(R)} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, $E_{f(R)} = \{e_1, e_2, e_3\}$ and:

- $\mathcal{L}_{f(R)}(n_1) = \{(\text{Interface}, \{p_1\}), (\text{ent}, J. \text{ Dorian}), (\text{type}, P), (\text{viewLabel}, J. \text{ Dorian})\};$
- $\mathcal{L}_{f(R)}(n_2) = \{(\text{Interface}, \{p_2, p_3, p_4\}), (\text{ent}, \text{Dr}(J. \text{ Lewis})), (\text{type}, C), (\text{viewLabel}, \text{Dr}(J. \text{ Lewis}))\};$
- $\mathcal{L}_{f(R)}(n_3) = \{(\text{Interface}, \{p_5, p_6, p_7\}), (\text{ent}, \text{Doctor}), (\text{type}, C), (\text{viewLabel}, \text{Doctor})\};$
- $\mathcal{L}_{f(R)}(p_1) = \{(\text{Attach}, n_1), (\text{Arity}, 2)\};$
- $\mathcal{L}_{f(R)}(p_2) = \{(\text{Attach}, n_2), (\text{Arity}, 1)\};$
- $\mathcal{L}_{f(R)}(p_3) = \{(\text{Attach}, n_2), (\text{Arity}, 0), (\text{viewLabel}, \text{In})\};$
- $\mathcal{L}_{f(R)}(p_4) = \{(\text{Attach}, n_2), (\text{Arity}, 1), (\text{viewLabel}, \text{Out})\};$
- $\mathcal{L}_{f(R)}(p_5) = \{(\text{Attach}, n_3), (\text{Arity}, 1)\};$
- $\mathcal{L}_{f(R)}(p_6) = \{(\text{Attach}, n_3), (\text{Arity}, 1), (\text{viewLabel}, \text{In})\};$
- $\mathcal{L}_{f(R)}(p_7) = \{(\text{Attach}, n_3), (\text{Arity}, 0), (\text{viewLabel}, \text{Out})\};$
- $\mathcal{L}_{f(R)}(e_1) = \{(\text{Connect}, \{p_1, p_2\}), (\text{type}, PC)\};$
- $\mathcal{L}_{f(R)}(e_2) = \{(\text{Connect}, \{p_4, p_6\}), (\text{type}, CC)\};$
- $\mathcal{L}_{f(R)}(e_3) = \{(\text{Connect}, \{p_1, p_5\}), (\text{type}, PC)\}.$

2. the rewriting phase: as $G \setminus f(L)$ is empty, there are no edges to connect to $f(R)$;

3. the deletion phase: $f(L)$ is erased obtaining G' which, in this case, is equal to $f(R)$.

2.4.3 Strategy Driven Rewriting

Given a set of rewrite rules and a graph, it will be possible to apply the rules in different orders and places of the graph. To control that, PORGY uses a strategy language. To define these strategies we start by defining the notions of position and banned subgraphs of a located graph.

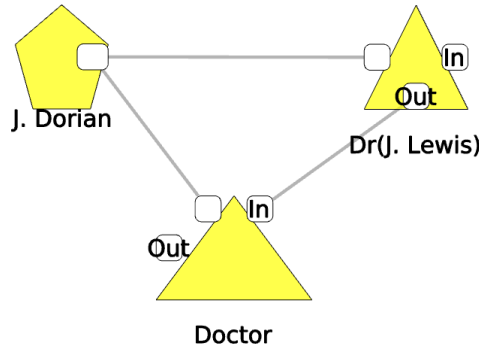


FIGURE 2.4: Result of a rewriting step

Definition 2.38 (Located graph). A located graph G_P^Q is a graph such that $P \subseteq G$ is called the position subgraph and $Q \subseteq G$ is the banned subgraph.

This notion is used in PORGY to select nodes, ports or edges where rewriting should happen and where it cannot happen.

Example 2.7. In the port graph in Figure 2.5 there are four possible applications of the rewrite rule shown in Figure 2.3, but if we define that nodes P. Cox and C. Tuck belong to the banned subgraph then there is only one possible application of the rule.

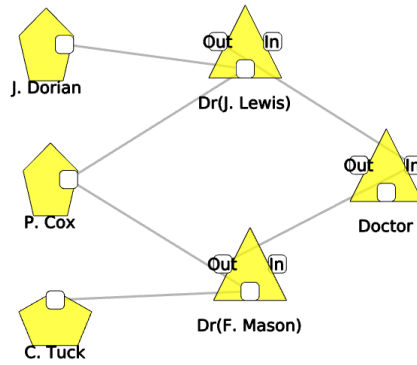


FIGURE 2.5: Located port graph

Definition 2.39 (Located rewrite rule). A located rewrite rule $L_W \Rightarrow R_M^N$ can be applied to a located graph G_P^Q by means of a morphism f if:

- $f(L) \cap Q = \emptyset$;
- $f(L) \cap P = f(W)$ or simply $f(L) \cap P \neq \emptyset$ if W is not specified;

in this case we write $G_P^Q \xrightarrow{f}_{L_W \Rightarrow R_M^N} G_{P'}^{Q'}$ where $P' = (P \setminus f(L)) \cup f(M)$ and $Q' = Q \cup f(N)$.

In lay terms W defines the elements of L which images should be in the position graph and M (resp. N) defines the elements of R which images are to be added to the position (resp. banned) subgraph of G' .

Example 2.8. In the rule shown in Figure 2.3 the top two nodes of L belong to W , the top two nodes of R belong to M and the bottom one belongs to N . This will allow us to control the rewriting using the position and banned subgraphs.

In PORGY we can use the strategy language (shown in Table 2.1) to control both the way rewrite rules are applied and the position and banned subgraphs. There are three main types of constructs. In the following we explain their usage.

Let L, R be port graphs; M, N subgraphs of R ; W a subgraph of L ; $n \in \mathbb{N}$; $\pi_{i=1\dots n} \in [0, 1]$, $\sum_{i=1}^n \pi_i = 1$; let <i>attribute</i> be an attribute label in Δ^A ; $e \in \mathcal{E}_\Delta$ a valid expression without variables			
Rules	Transformations	T	$::= L_W \Rightarrow R_M^N \mid T \parallel T$ $\mid \text{ppick}(T_1, \pi_1, \dots, T_n, \pi_n)$
	Applications	A	$::= \text{one}(T) \mid \text{all}(T)$
Positions	Focusing	F	$::= \text{crtGraph} \mid \text{crtPos} \mid \text{crtBan} \mid [\text{emptySet}]$ $\mid F[\text{cup}]F \mid F[\text{cap}]F \mid F \setminus F \mid (F)$ $\mid \text{ppick}(F_1, \pi_1, \dots, F_n, \pi_n)$ $\mid \text{property}(F, \rho) \mid \text{nbg}(F, \rho)$
	Determine	D	$::= \text{all}(F) \mid \text{one}(F)$
	Update	U	$::= \text{setPos}(D) \mid \text{setBan}(D)$
Properties	Properties	ρ	$::= \text{Elem}, \text{Expr}$
		Elem	$::= \text{node} \mid \text{edge} \mid \text{port}$
		Expr	$::= \text{attribute} \text{ Relop } e \mid \text{true}$
		Relop	$::= = \mid ! = \mid > \mid < \mid > = \mid < = \mid \sim$
Compositions	Comparison	C	$::= F = F \mid F \neq F \mid F \subset F$ $\mid \text{isEmpty}(F) \mid \text{match}(T)$
	Strategies	S	$::= \text{id} \mid \text{fail} \mid A \mid U \mid C \mid S; S$ $\mid \text{if}(S)\text{then}(S)[\text{else}(S)] \mid (S)\text{orelse}(S)$ $\mid \text{repeat}(S)[(n)] \mid \text{while}(S)\text{do}(S)[(n)]$ $\mid \text{try}(S) \mid \text{not}(S)$

TABLE 2.1: Syntax of strategy language

- **Rule Constructs** are the way to apply located rewrite rules to the graph, $T \parallel T$ denotes the application of both rules concurrently in disjoint parts of the graph, construct $\text{ppick}(T_1, \pi_1, \dots, T_n, \pi_n)$, where $\sum_{i=1}^n \pi_i = 1$, defines that T_i will be applied

with probability π_i , also that $T_i || T_j$ will be applied with probability $\pi_i \times \pi_j$. Construct $\text{all}(T)$ means that every possible application of T is going to be performed which produces as many children of the initial graph node as there are possible applications of T , $\text{one}(T)$ chooses one of the possible applications at random, both of them return fail if there are no possible applications.

- **Position Constructs** are divided into three categories: **Focusing** expressions are used to get subgraphs of a given graph; **Determine** expressions use focusing ones to return either one (chosen at random) or all the nodes of the given graph; and **Update** expressions are the way to set the position and banned subgraphs.

- **Focusing constructs**

- * $\text{crtGraph}, \text{crtPos}, \text{crtBan}$ return the whole graph, the position subgraph and the banned subgraph respectively;
- * $F[\text{cup}]F, F[\text{cap}]F, F \setminus F$ define respectively the graph operations union, intersection and complement, where $[\text{emptySet}]$ represents the empty graph;
- * $\text{ppick}(F_1, \pi_1, \dots, F_n, \pi_n)$ returns the graph defined by the expression F_i with probability π_i ;
- * $\text{property}(F, \rho)$ returns the elements of the graph defined by the expression F that satisfy the property ρ ;
- * $\text{nbg}(F, \rho)$ returns the neighbors of the graph defined by F that are connected to it by an element that satisfies the property ρ .

- **Determine constructs**

- * $\text{all}(F)$ returns all the nodes of the subgraph defined by F , while $\text{one}(F)$ returns only one of them chosen at random.

- **Update constructs**

- * $\text{setPos}(D)$ (resp. $\text{setBan}(D)$) sets the position (resp. banned) subgraph to the one defined by the expression D .

- **Composition Constructs** are divided into two types of expressions: **Comparison** ones that return only fail or id, which are two atomic strategies to denote failure or success respectively; and **Strategies** that control the flow of execution.

– **Comparison constructs**

- * $F = F$ returns id if both expressions define the same subgraph and fail otherwise;
- * $F \neq F$ returns id if both expressions do not define the same subgraph and fail otherwise;
- * $F \subset F$ returns id if the subgraph defined by the left expression is a subset of the one defined by the right expression and fail otherwise;
- * $\text{isEmpty}(F)$ returns id if the expression F defines the empty graph and fail otherwise;
- * $\text{match}(T)$ returns id if the located rewrite rule T can be applied in the graph and fail otherwise.

– **Strategy constructs**

- * $S_1; S_2$ defines sequential application of S_1 and S_2 ;
- * $\text{if}(S_1)\text{then}(S_2)[\text{else}(S_3)]$ tries to apply S_1 to a copy of the graph, if it succeeds then it applies S_2 to the graph, otherwise it applies S_3 . If S_3 is not specified it is considered to be id;
- * $(S_1)\text{orelse}(S_2)$ applies S_1 unless it fails, in which case it applies S_2 ;
- * $\text{repeat}(S)[(n)]$ applies S until it fails or the number of applications reaches n (if specified);
- * $\text{while}(S_1)\text{do}(S_2)[(n)]$ applies S_2 until the application of S_1 in a copy of the graph fails or the number of applications reaches n (if specified);
- * $\text{try}(S)$ applies S if the application does not fail, it is the same as using $(S)\text{orelse}(\text{id})$;
- * $\text{not}(S)$ returns id if the application of S fails and fail otherwise.

Example 2.9. To use the rule shown in Figure 2.3 called *auxPC*, in the graph shown in Figure 2.5 we use the following strategy, with the result being the graph in Figure 2.6. What this strategy does is, for every node of type P , create the new edges of type PC that come from the existing ones and repeat the process for the newly created ones until there are none left to create.

```

setBan(all(property(crtGraph,node,type=="P")));
while(not(isEmpty(crtBan)))do(
  setPos(one(crtBan));
  setBan(all(crtBan\crtPos));
  setPos(all(crtPos[cup]ngb(crtPos,edge,type=="PC")));
  while(one(auxPC))do(
    repeat(one(auxPC));
    setPos(all(property(crtPos,node,type=="P") [cup]
      property(crtBan,node,type=="C")));
    setBan(all(property(crtBan,node,type=="P")))
  ))

```

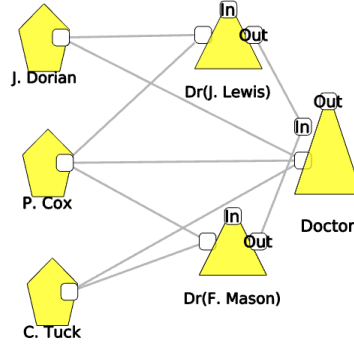


FIGURE 2.6: Result of port graph rewriting using strategies

To clarify how the strategy works let's follow it step by step.

1. the first *setBan* puts nodes J. Dorian, P. Cox and C. Tuck in the banned subgraph;
2. because *crtBan* is not empty it enters the first while cycle;
 - (a) *setPos* sets the position subgraph to be one of the nodes in *crtBan* (in this case J. Dorian);
 - (b) *setBan* takes the node added to the position subgraph out of the banned subgraph (now composed of nodes P. Cox and C. Tuck);
 - (c) *setPos* adds nodes connected to the node in *crtPos* by edges of type PC to the position subgraph (now composed of nodes J. Dorian and Dr(J. Lewis));
 - (d) because rewrite rule *auxPC* can be applied it enters the second while cycle;

- i. rewrite rule *auxPC* is applied resulting in the graph shown in Figure 2.7 where node Doctor was added to the banned subgraph (now composed of nodes P. Cox, C. Tuck and Doctor);
 - ii. rewrite rule *auxPC* cannot be applied again;
 - iii. *setPos* sets the position subgraph to be composed of the nodes of type P already in *crtPos* (J. Dorian) and the nodes of type C in *crtBan* (Doctor);
 - iv. *setBan* sets the banned subgraph to be composed of the nodes of type P already in *crtBan* (P. Cox and C. Tuck);
- (e) because rewrite rule *auxPC* cannot be applied it exits the second while cycle;
- 3. because *crtBan* is not empty it continues in the first while cycle;
 - (a) *setPos* sets the position subgraph to be one of the nodes in *crtBan* (in this case C. Tuck);
 - (b) *setBan* takes the node added to the position subgraph out of the banned subgraph (now composed of node P. Cox);
 - (c) *setPos* adds nodes connected to the node in *crtPos* by edges of type PC to the position subgraph (now composed of nodes C. Tuck and Dr(F. Mason));
 - (d) because rewrite rule *auxPC* can be applied it enters the second while cycle;
 - i. rewrite rule *auxPC* is applied resulting in the graph shown in Figure 2.8 where node Doctor was added to the banned subgraph (now composed of nodes P. Cox and Doctor);
 - ii. rewrite rule *auxPC* cannot be applied again;
 - iii. *setPos* sets the position subgraph to be composed of the nodes of type P already in *crtPos* (C. Tuck) and the nodes of type C in *crtBan* (Doctor);
 - iv. *setBan* sets the banned subgraph to be composed of the nodes of type P already in *crtBan* (P. Cox);
 - (e) because rewrite rule *auxPC* cannot be applied it exits the second while cycle;
- 4. because *crtBan* is not empty it continues in the first while cycle;
 - (a) *setPos* sets the position subgraph to be one of the nodes in *crtBan* (the only one left is P. Cox);
 - (b) *setBan* takes the node added to the position subgraph out of the banned subgraph (now empty);

- (c) *setPos* adds nodes connected to the node in *crtPos* by edges of type *PC* to the position subgraph (now composed of nodes *P. Cox*, *Dr(J. Lewis)* and *Dr(F. Mason)*);
- (d) because rewrite rule *auxPC* can be applied it enters the second while cycle;
 - i. rewrite rule *auxPC* is applied resulting in the graph shown in Figure 2.6 where node *Doctor* was added to the banned subgraph (now composed of node *Doctor*);
 - ii. rewrite rule *auxPC* cannot be applied again;
 - iii. *setPos* sets the position subgraph to be composed of the nodes of type *P* already in *crtPos* (*P. Cox*) and the nodes of type *C* in *crtBan* (*Doctor*);
 - iv. *setBan* sets the banned subgraph to be composed of the nodes of type *P* already in *crtBan* (which in this case is none making the banned subgraph empty);
- (e) because rewrite rule *auxPC* cannot be applied it exits the second while cycle;
- 5. because *crtBan* is empty it exits the first while cycle.

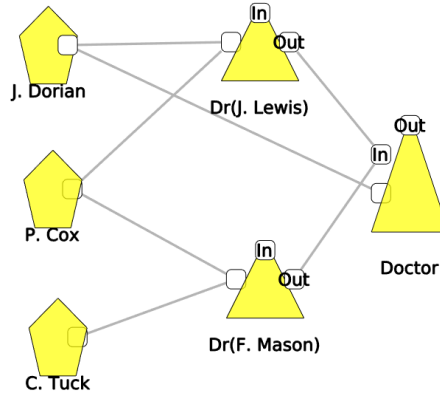


FIGURE 2.7: Result of port graph rewriting using strategies (first step)

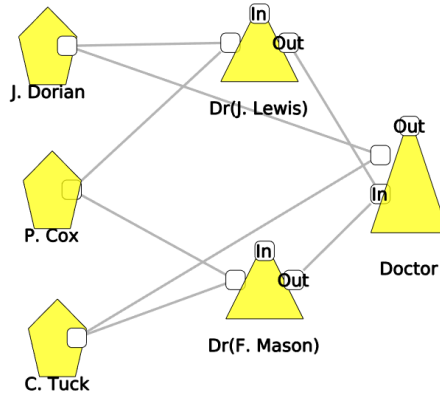


FIGURE 2.8: Result of port graph rewriting using strategies (second step)

Now that the reader is acquainted with the notions of CBAC and CBACO, the graph-based framework for representing CBAC policies and the PORGY tool, we are ready to present our work. Starting with the next chapter, where we introduce our extension of the framework for the representation of CBAC policies to CBACO.

Chapter 3

Graphs for dynamic CBACO policies

In the previous chapter we have discussed the CBAC and CBACO models and presented a graph representation for CBAC policies. In this chapter we present a new graphical model to represent dynamic CBACO policies in which the actual policy is represented by the graph and the possible dynamism of the policy is represented by means of rewrite rules. We will start by focusing on the static representation and then on the dynamics. But before we get to the graphic representation of CBACO policies, let us first introduce to CBACO a set of new relations that deal with the state of duties.

3.1 State of Duties

As stated in the previous chapter, issued obligations (also called duties), can be in one of four states: invalid, fulfilled, pending or violated. This is unlike permissions, which are either issued or not. To deal with that, we define the following relations for the status of duties:

- Fulfilled: $FULFILLEDD \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{H}$, such that $(d, e, h) \in FULFILLEDD$ if the duty d was fulfilled by the event e in history h .
- Pending: $PENDING \subseteq \mathcal{D} \times \mathcal{H}$, such that $(d, h) \in PENDING$ if the duty d is pending in history h .
- Violated: $VIOLATED \subseteq \mathcal{D} \times \mathcal{H}$, such that $(d, h) \in VIOLATED$ if the duty is violated in history h .

We do not consider a relation on invalid duties, since our model does not allow them to be issued. Furthermore, we introduce the following axioms regarding the state of duties.

$$\begin{aligned}
& \forall \overbrace{(p, a, r, e_1, e_2)}^d \in \mathcal{DA}, \forall h \in \mathcal{H}((e_1 \in h, \\
& \quad \exists e_3 \in h, (e_1, e_3, h) \in \mathcal{EI}, ((e_3, e_2, h) \in \mathcal{EI} \vee (e_1, e_2, h) \notin \mathcal{EI}), \\
& \quad e_3.\text{subj} = p, e_3.\text{act} = a, e_3.\text{obj} = r) \Leftrightarrow \\
& \quad (d, e_3, h) \in \mathcal{FULFILLED} \quad (3.1)
\end{aligned}$$

$$\begin{aligned}
& \forall \overbrace{(p, a, r, e_1, e_2)}^d \in \mathcal{DA}, \forall h \in \mathcal{H}((e_1 \in h, (e_1, e_2, h) \notin \mathcal{EI}, \\
& \quad \nexists e_3 \in h, (e_1, e_3, h) \in \mathcal{EI}, e_3.\text{subj} = p, e_3.\text{act} = a, e_3.\text{obj} = r) \Leftrightarrow \\
& \quad (d, h) \in \mathcal{PENDING} \quad (3.2)
\end{aligned}$$

$$\begin{aligned}
& \forall \overbrace{(p, a, r, e_1, e_2)}^d \in \mathcal{DA}, \forall h \in \mathcal{H}((e_1, e_2, h) \in \mathcal{EI}, \\
& \quad \nexists e_3 \in (e_1, e_2, h), e_3.\text{subj} = p, e_3.\text{act} = a, e_3.\text{obj} = r) \Leftrightarrow \\
& \quad (d, h) \in \mathcal{VIOLATED} \quad (3.3)
\end{aligned}$$

3.2 Static CBACO policies

There are many things we have to add to Definition 2.19 to define the representation of CBACO policies. In addition to the necessary notions to describe the new relations, we will also make some changes in the way some of the already described relations are represented. We follow by defining CBACO policy graphs and then justify the changes that were made.

Definition 3.1 (CBACO policy graph). A CBACO policy graph G is a tuple $\langle N_G, E_G, ln, le \rangle$ with:

- N_G a finite set of nodes (denoted $n_1, n_2, \dots$);
- E_G a finite set of undirected edges (denoted $e_1, e_2, \dots$);
- $ln : N_G \rightarrow \mathcal{REC}$ a labeling function for nodes such that for every node $n \in N_G$, $ln(n).ent \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{A} \cup \mathcal{R} \cup \mathcal{G} \cup \mathcal{E} \cup (\mathcal{A} \times \mathcal{R}) \cup (\mathcal{A} \times \mathcal{R} \times \mathcal{G} \times \mathcal{G}) \cup (\mathcal{P} \times \mathcal{A} \times \mathcal{R} \times \mathcal{E} \times \mathcal{E})$ and $ln(n).type \in \{P, C, A, R, G, E, Pr, O, D\}$ where:

- $ln(n).type = P$ iff $ln(n).ent = p \in \mathcal{P}$;
 - $ln(n).type = C$ iff $ln(n).ent = c \in \mathcal{C}$;
 - $ln(n).type = A$ iff $ln(n).ent = a \in \mathcal{A}$;
 - $ln(n).type = R$ iff $ln(n).ent = r \in \mathcal{R}$;
 - $ln(n).type = G$ iff $ln(n).ent = ge \in \mathcal{G}$;
 - $ln(n).type = E$ iff $ln(n).ent = e \in \mathcal{E}$;
 - $ln(n).type = P_r$ iff $ln(n).ent = (a, r) \in \mathcal{A} \times \mathcal{R}$;
 - $ln(n).type = O$ iff $ln(n).ent = (a, r, ge_1, ge_2) \in \mathcal{A} \times \mathcal{R} \times \mathcal{G} \times \mathcal{G}$;
 - $ln(n).type = D$ iff $ln(n).ent = (p, a, r, e_1, e_2) \in \mathcal{P} \times \mathcal{A} \times \mathcal{R} \times \mathcal{E} \times \mathcal{E}$;
- $le : E_G \rightarrow \mathcal{REC}$ a labeling function for edges such that for every edge $e \in E_G$, $le(e).adj = \{n_1, n_2\}$ gives the two nodes connected by the edge. Function $type$ is defined as in Definition 2.19, furthermore if:
- $type(e) = CC$ then $le(e).target \subseteq \{ln(n_1).ent, ln(n_2).ent\}$ and $le(e).auth, le(e).obl \in \{\top, \perp\}$;
 - $type(e) = CP_r$ then $le(e).auth \in \{A, B\}$;
 - $type(e) = OG$ then $le(e).ge \in \{i, f\}$;
 - $type(e) = DE$ then $le(e).ev \in \{i, f\}$;
 - $type(e) = GG$ then $le(e).target \in \{ln(n_1).ent, ln(n_2).ent\}$;
 - $type(e) = EE$ then $le(e).target \in \{ln(n_1).ent, ln(n_2).ent\}$;

The addition of nodes of type G, E to represent entities of $\mathcal{G}, \mathcal{E}$, respectively, is straightforward. Nodes of type E have attribute *now* set to either $\top$ or $\perp$. Only one node of type E can have *now* set to $\top$, which specifies what is the current event.

In Definition 2.19 nodes that represent actions have a field defining to which records they correspond. This is needed because categories are connected to actions and if there was only one node for each action then the policy could not be well represented. What this means is that the same action could be represented by different nodes which is not good for the expressiveness of the representation neither for the use of rewrite rules. By using nodes of type P_r to represent pairs action/resource we no longer have the need to use different nodes to represent the same action. These nodes are always connected to a node of type A and one of type R , the pair it represents, and can be connected to nodes of

type C to represent a permission/prohibition or to nodes of type O (resp. D) to represent the pair action/resource of an obligation (resp. duty).

Nodes of type O represent obligations and are always connected to a node of type P_r and zero, one or two nodes of type G . Other connections will be to nodes of type C .

Nodes of type D represent duties and are always connected to a node of type P a node of type P_r and zero, one or two nodes of type E .

Because we represent both $\subseteq$ and $\subseteq_O$ there is the need to distinguish edges of type CC . Edges e that represent the $\subseteq$ (resp. $\subseteq_O$) will have the *auth* (resp. *obl*) attribute value set to $\top$ and we write $type(e) = CC_{P_r}$ (resp. $type(e) = CC_O$).

Edges e of type CP_r serve the same purpose of the edges of type CA in Definition 2.28 and for that reason have attribute *auth* defined as either A or B and we will write $type(e) = CP_r^A$ and $type(e) = CP_r^B$, respectively, to distinguish them.

Edges of type OG (resp. DE) have the *ge* (resp. *ev*) attribute value set to either i or f to distinguish between the initial and final event scheme (resp. event).

Edges of type GG are used to represent the notion that some event schemes are instantiations of other event schemes and have attribute *target* set to the node that represents the event scheme that is more general (the one which the other is an instantiation of). Furthermore, the type of this edges when in terms of types of paths will have an arrow on top in the same way edges of type CC have.

Edges of type EE connect events of the same history and have attribute *target* set to the node that represents the one event that comes after the other. As edges of type CC and GG the type of this edges in paths will have an arrow on top.

Definition 3.2 (Redundant edges in the CBACO policy graph). Consider a CBACO policy graph $G = \langle N_G, E_G, ln, le \rangle$. An edge $e \in E_G$ is redundant if:

- $type(e) = PC$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = p$ and $ln(n_2).ent = c$ with $p \in \mathcal{P}, c \in \mathcal{C}$, and there is a path of type $PC, (\overrightarrow{CC})^*$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = \overrightarrow{CC}_{P_r}$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c_1$ and $ln(n_2).ent = c_2$ with $c_1, c_2 \in \mathcal{C}$ and $le(e).auth = \top$ and $c_2 \in le(e).target$, and there is a path of type $(\overrightarrow{CC}_{P_r})^*$ and length greater or equal to 2 connecting n_1 and n_2 ;

- $type(e) = \overrightarrow{CC}_O$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c_1$ and $ln(n_2).ent = c_2$ with $c_1, c_2 \in \mathcal{C}$ and $le(e).obl = \top$ and $c_2 \in le(e).target$, and there is a path of type $(\overrightarrow{CC}_O)^*$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = CP_r^A$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c$ and $ln(n_2).ent = (a, r)$ with $c \in \mathcal{C}$, $(a, r) \in \mathcal{A} \times \mathcal{R}$ and $le(e).auth = A$, and there is a path of type $(\overrightarrow{CC}_{P_r})^*, CP_r^A$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = CP_r^B$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c$ and $ln(n_2).ent = (a, r)$ with $c \in \mathcal{C}$, $(a, r) \in \mathcal{A} \times \mathcal{R}$ and $le(e).auth = B$, and there is a path of type $(\overleftarrow{CC}_{P_r})^*, CP_r^B$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = CO$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = c$ and $ln(n_2).ent = (a, r, ge_1, ge_2)$ with $c \in \mathcal{C}$, $(a, r, ge_1, ge_2) \in \mathcal{A} \times \mathcal{R} \times \mathcal{G} \times \mathcal{G}$, and there is a path of type $(\overrightarrow{CC}_O)^*, CO$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = EG$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = e$ and $ln(n_2).ent = ge$ with $e \in \mathcal{E}$, $ge \in \mathcal{G}$, and there is a path of type $EG, (\overrightarrow{GG})^*$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = \overrightarrow{GG}$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = ge_1$ and $ln(n_2).ent = ge_2$ with $ge_1, ge_2 \in \mathcal{G}$ and $ge_2 = le(e).target$, and there is a path of type $(\overrightarrow{GG})^*$ and length greater or equal to 2 connecting n_1 and n_2 ;
- $type(e) = \overrightarrow{EE}$, i.e. it connects nodes n_1, n_2 such that $ln(n_1).ent = e_1$ and $ln(n_2).ent = e_2$ with $e_1, e_2 \in \mathcal{E}$ and $e_2 = le(e).target$, and there is a path of type $(\overrightarrow{EE})^*$ and length greater or equal to 2 connecting n_1 and n_2 .

The notions of constrained path and constrained inverse path are the same with the addition that edges of the path must have attribute *auth* value set to $\top$ when it is specified, meaning they only apply to authorization paths.

Definition 3.3 (Well-formed CBACO policy graph). A CBACO policy graph G is well-formed iff for every $n_1, n_2 \in N_G$ if $ln(n_1).ent = ln(n_2).ent$ and $ln(n_1).type = ln(n_2).type$ then $n_1 = n_2$, for every $e_1, e_2 \in E_G$ if $le(e_1).adj = le(e_2).adj$ then either $e_1 = e_2$ or e_1, e_2 have type CP_r and $le(e_1).auth \neq le(e_2).auth$, for every $e \in E_G$ with $le(e).adj = \{n_1, n_2\}$ one of the following is true:

- $ln(n_1).type = P \wedge ln(n_2).type = C$;

- $\ln(n_1).type = C \wedge \ln(n_2).type = C \wedge le(e).target \subseteq \{\ln(n_1).ent, \ln(n_2).ent\} \wedge le(e).auth, le(e).obl \in \{\top, \perp\}$;
- $\ln(n_1).type = C \wedge \ln(n_2).type = P_r \wedge le(e).auth \in \{A, B\}$;
- $\ln(n_1).type = C \wedge \ln(n_2).type = O$;
- $\ln(n_1).type = P_r \wedge \ln(n_2).type = A \wedge \ln(n_1).ent = (a, r) \wedge \ln(n_2).ent = a$;
- $\ln(n_1).type = P_r \wedge \ln(n_2).type = R \wedge \ln(n_1).ent = (a, r) \wedge \ln(n_2).ent = r$;
- $\ln(n_1).type = O \wedge \ln(n_2).type = P_r \wedge \ln(n_2).ent = (a, r) \wedge \ln(n_1).ent = (a, r, ge_1, ge_2)$;
- $\ln(n_1).type = O \wedge \ln(n_2).type = G \wedge \ln(n_1).ent = (a, r, ge_1, ge_2) \wedge ((le(e).ge = i \wedge \ln(n_2).ent = ge_1) \vee (le(e).ge = f \wedge \ln(n_2).ent = ge_2))$;
- $\ln(n_1).type = D \wedge \ln(n_2).type = \mathcal{P} \wedge \ln(n_2).ent = p \wedge \ln(n_1).ent = (p, a, r, e_1, e_2)$;
- $\ln(n_1).type = D \wedge \ln(n_2).type = P_r \wedge \ln(n_2).ent = (a, r) \wedge \ln(n_1).ent = (p, a, r, e_1, e_2)$;
- $\ln(n_1).type = D \wedge \ln(n_2).type = E \wedge \ln(n_1).ent = (p, a, r, e_1, e_2) \wedge ((le(e).ev = i \wedge \ln(n_2).ent = e_1) \vee (le(e).ev = f \wedge \ln(n_2).ent = e_2))$;
- $\ln(n_1).type = E \wedge \ln(n_2).type = E \wedge le(e).target \in \{\ln(n_1).ent, \ln(n_2).ent\}$;
- $\ln(n_1).type = E \wedge \ln(n_2).type = P$;
- $\ln(n_1).type = E \wedge \ln(n_2).type = A$;
- $\ln(n_1).type = E \wedge \ln(n_2).type = R$;
- $\ln(n_1).type = E \wedge \ln(n_2).type = G$;
- $\ln(n_1).type = G \wedge \ln(n_2).type = G \wedge le(e).target \in \{\ln(n_1).ent, \ln(n_2).ent\}$;

and there are no redundant edges. Moreover if a constrained path and an inverse constrained path begin in the same node with type P and end in nodes of type P_r such that the last edges of the paths are of type CP_r^A and CP_r^B , respectively, then the end node must be different. This means that a principal cannot be both able and unable to perform an action on a resource.

Definition 3.4 (CBACO_G). Given a well-formed CBACO policy graph $G = \langle N_G, E_G, \ln, le \rangle$ the tuple $\langle \Gamma_G, PCA_G, ARCA_G, PAR_G, BARCA_G, BAR_G, UNDET_G, OCA_G, OPA_G, \dots \rangle$

$\mathcal{DA}_G, \mathcal{ET}_G, \mathcal{EI}_G\rangle$, denoted CBACO_G , where $\Gamma_G = \langle \mathcal{P}_G, \mathcal{C}_G, \mathcal{A}_G, \mathcal{R}_G, \subseteq, \subseteq_{\mathcal{O}}, \mathcal{E}_G, \mathcal{G}_G, \mathcal{H}_G \rangle$ and:

- $\mathcal{P}_G = \{ln(n).ent \mid ln(n).type = P\};$
- $\mathcal{C}_G = \{ln(n).ent \mid ln(n).type = C\};$
- $\mathcal{A}_G = \{ln(n).ent \mid ln(n).type = A\};$
- $\mathcal{R}_G = \{ln(n).ent \mid ln(n).type = R\};$
- $\subseteq = \{(ln(n_1).ent, ln(n_i).ent) \mid \exists n_2, \dots, n_{i-1}, type(n_1, \dots, n_i) = (\overrightarrow{CC_P})^*\};$
- $\subseteq_{\mathcal{O}} = \{(ln(n_1).ent, ln(n_i).ent) \mid \exists n_2, \dots, n_{i-1}, type(n_1, \dots, n_i) = (\overrightarrow{CC_O})^*\};$
- $\mathcal{E}_G = \{ln(n).ent \mid ln(n).type = E\};$
- $\mathcal{G}_G = \{ln(n).ent \mid ln(n).type = G\};$
- $\mathcal{H}_G = \{[ln(n_1).ent, \dots, ln(n_i).ent] \mid type(n_1, \dots, n_i) = (\overrightarrow{EE})^*\};$
- $\mathcal{PCA}_G = \{(ln(n_1).ent, ln(n_2).ent) \mid type(n_1, n_2) = PC\};$
- $\mathcal{ARCA}_G = \{(a, r, ln(n_1).ent) \mid \exists n_2, type(n_1, n_2) = CP_r^A \wedge ln(n_2).ent = (a, r)\};$
- $\mathcal{PAR}_G = \{(ln(n_1).ent, a, r) \mid \exists n_{21}, \dots, n_{2i}, n_3,$
 $type(n_1, n_{21}, \dots, n_{2i}, n_3) = PC, (\overrightarrow{CC_P})^*, CP_r^A \wedge ln(n_3).ent = (a, r)\};$
- $\mathcal{BARCA}_G = \{(a, r, ln(n_1).ent) \mid \exists n_2, type(n_1, n_2) = CP_r^B \wedge ln(n_2).ent = (a, r)\};$
- $\mathcal{BAR}_G = \{(ln(n_1).ent, a, r) \mid \exists n_{21}, \dots, n_{2i}, n_3,$
 $type(n_1, n_{21}, \dots, n_{2i}, n_3) = PC, (\overleftarrow{CC_P})^*, CP_r^B \wedge ln(n_3).ent = (a, r)\};$
- $\mathcal{UNDET}_G = \{(ln(n_1).ent, ln(n_2).ent, ln(n_3).ent) \mid ln(n_1).type = P, ln(n_2).type = A, ln(n_3).type = R\} \setminus (\mathcal{PAR}_G \cup \mathcal{BAR}_G);$
- $\mathcal{OCA}_G = \{(a, r, ge_1, ge_2, ln(n_1).ent) \mid \exists n_2, type(n_1, n_2) = CO \wedge$
 $ln(n_2).ent = (a, r, ge_1, ge_2)\};$
- $\mathcal{OPA}_G = \{(ln(n_1).ent, a, r, ge_1, ge_2) \mid \exists n_{21}, \dots, n_{2i}, n_3,$
 $type(n_1, n_{21}, \dots, n_{2i}, n_3) = PC, (\overrightarrow{CC_O})^*, CO \wedge ln(n_3).ent = (a, r, ge_1, ge_2)\};$
- $\mathcal{DA}_G = \{ln(n).ent \mid ln(n).type = D\};$

- $\mathcal{ET}_G = \{(ln(n_1).ent, ln(n_i).ent) \mid \exists n_2, \dots, n_{i-1}, type(n_1, \dots, n_i) = EG, (\overrightarrow{GG})^*\};$
- $\mathcal{EI}_G = \{(ln(n_j).ent, ln(n_k).ent, [ln(n_1).ent, \dots, ln(n_i).ent]) \mid type(n_1, \dots, n_i) = (\overrightarrow{EE})^* \wedge 1 \leq j < k \leq i\};$

defines the CBACO policy described by graph G .

3.3 Dynamic CBACO policies

As already stated the dynamics of the policy will be represented by means of rewrite rules. We can separate these rules in two, the rules that are general to every CBACO policy and will not change the policy itself (the ones associated with queries and obligations), and the rules that are scenario specific and will change the policy. This will only be a textual introduction to the purpose of the rules and why they are needed, examples of the actual rules will be shown in Chapter 5.

3.3.1 Scenario-specific rewrite rules

This category covers a wide variety of rules that we cannot fully predict because of the fact that they are dependant on the policy. Their purpose is to represent the dynamics specific to the policy: ways of creating/removing entities, updating relations or constraints of the policy. They can be triggered either by events or by specific states of the policy. In the example shown in Section 5.1 the $\mathcal{PCA}$ relation is updated based on the attributes of the principals.

3.3.2 General rewrite rules

In this subsection we focus on defining rewrite rules that are general to every CBACO policy. Inside the scope of these rewrite rules we will focus first on auxiliary ones that are needed for some of the following to work, then on the ones related to obligations and then on rules that give response to queries.

3.3.2.1 Auxiliary rewrite rules

As stated above these rules are needed for the following (and possibly the scenario-specific ones) to work. Their purpose is to make the relations that are implicit in the graph explicit. For example, consider we have principal p connected to category c and c

connected to category c' by a path of length n and type $(\vec{CC}_{P_r})^*$, i.e. $c \subseteq c'$, and we want to show the $\mathcal{PAR}$ relation. We know that, by Axiom 2.1, principal p will inherit the permissions assigned to c and c' , but if we wanted to create rules that directly expressed this we would have to create a different rule for each possible path length connecting categories c and c' . There are different ways of solving this issue, the one we implemented that is shown in Section 5.2 is to create auxiliary edges connecting principals and categories that are in this type of relation and using said edges in the following rules.

3.3.2.2 Obligation rewrite rules

Obligation rewrite rules will concern the dynamics of duties and its states. In this category of rules we have rules to instantiate the two kinds of duty that need different instantiation rules (the ones associated with obligations where ge_1 is $\perp$ and the ones associated with obligations where ge_1 is some $ge \in G$) and rules for every possible update of the state of a duty. Also in this category we put the rules concerning the iteration of events, this is due to the fact that, although they could also be necessary for the scenario-specific rules, they are central to the calculation of duties and their states. All the rules in this category are shown in Section 5.3.

3.3.2.3 Visualization and queries rewrite rules

Queries rewrite rules concern possible queries that can be made by a system manager. They will rely on the rules mentioned above (auxiliary and obligation) to get the requested answer coupled with visualization rules to show said answer. Visualization rules are just rules to hide or show specific nodes or edges, based on their attributes, that can be used to get different views of the policy graph by using certain subsets of them. In Section 5.4 we show two examples (one for nodes and one for edges) of visualization rules to hide/show elements of the graph, how to use them and other visualization rules to get views of the graph and how to use them coupled with auxiliary rules to show the $\mathcal{UNDET}$ relation.

Chapter 4

Representing CBAC(O) in PORGY

In this chapter we will describe our implementation of the static part of the CBACO representation framework introduced in the previous chapter using PORGY. This will be done in two parts, first we describe the part of the representation that concerns only permissions/prohibitions and then we describe the part that concerns obligations. In this description we mention only the essential attributes, the ones inherent to the model. Both nodes and edges can have additional attributes depending on the specific policy being represented.

4.1 Graphical CBAC in PORGY

We start by describing our choices regarding the representation of the entities and the relations in the policies.

4.1.1 Entities

Entities are represented as nodes of the graph, their *viewLabel* will be the same as *ent* and the shape will be determined by the *type*. The number of ports is also determined by the *type* of the entity. With the exception of edges representing the $\subseteq$ relation between categories, every other edge is undirected. Therefore, nodes of type C will have three ports while the others only have one. The extra two ports (which have *viewLabel* **In** and **Out**) on category nodes are used to make the $\subseteq$ relation visually explicit. These properties are summarized in Table 4.1.

<i>type</i>	<i>viewShape</i>	<i>number of ports</i>
P	Pentagon	1
C	Triangle	3
P_r	Hexagon	1
A	Square	1
R	Diamond	1

TABLE 4.1: Properties of nodes in a CBAC representation

4.1.2 Relations

Relations are represented as edges of the graph. The *viewColor* of edges is gray except for edges of type CP_r , which are green if its *auth* is A and red if it is B . Edges of type CC will connect to the extra ports of nodes with *type* C , connecting to the **In** port if the *ent* of the node is in the *target* of the edge or to the **Out** port otherwise.

Example 4.1. In Figure 4.1 we have the representation of a CBAC policy such that:

- $\mathcal{P} = \{C. Tuck\};$
- $\mathcal{C} = \{Dr(F. Mason), Doctor\};$
- $\mathcal{A} = \{Read, Perform\};$
- $\mathcal{R} = \{Rota, Rec(F. Mason), Specimen collection\};$
- $\mathcal{PCA} = \{(C. Tuck, Dr(F. Mason))\};$
- $\subseteq = \{(Dr(F. Mason), Doctor)\};$
- $\mathcal{ARCA} = \{(Read, Rota, Doctor), (Read, Rec(F. Mason), Dr(F. Mason))\};$
- $\mathcal{PAR} = \{(C. Tuck, Read, Rota), (C. Tuck, Read, Rec(F. Mason))\};$
- $\mathcal{BARCA} = \{(Perform, Specimen collection, Doctor)\};$
- $\mathcal{BAR} = \emptyset;$
- $\mathcal{UNDET} = \{(C. Tuck, Perform, Rota), (C. Tuck, Perform, Rec(F. Mason)),$
 $(C. Tuck, Perform, Specimen collection), (C. Tuck, Read, Specimen collection), \}$.

Note that: the edge connecting the two category nodes (triangles) connects to the **Out** port of $Dr(F. Mason)$ and the **In** port of $Doctor$, making clear that $Dr(F. Mason) \subseteq Doctor$; the edges representing the $\mathcal{ARCA}$ relation are green while the one representing the $\mathcal{BARCA}$ relation is red.

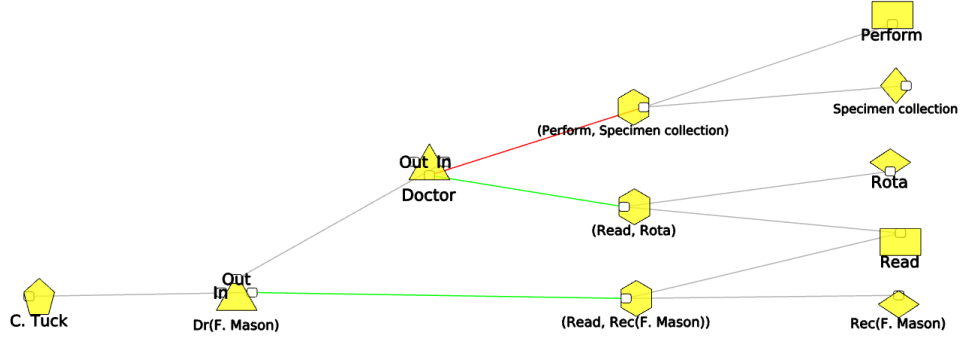


FIGURE 4.1: CBAC policy representation

4.2 Graphical CBACO in PORGY

To represent CBACO policies we will have new entities and relations. In order to make a visual distinction between the permissions/prohibitions and the obligations subgraphs we will use the color of nodes.

4.2.1 Entities related to obligations

The new entities we have to represent are obligations, duties, events and event schemes which will be represented as nodes of the graph. As in the CBAC part of the representation their *viewLabel* will be the same as *ent* and the shape and number of ports will be determined by the *type*. Obligation and duty nodes both have one port while event and event scheme nodes have three ports as category nodes. This extra ports are used to represent the direction of edges with *type* *EE* and *GG*. The properties of these new nodes are summarized in Table 4.2.

<i>type</i>	<i>viewShape</i>	<i>number of ports</i>
<i>O</i>	Hexagon	1
<i>D</i>	Hexagon	1
<i>E</i>	Circle	3
<i>G</i>	Ring	3

TABLE 4.2: Properties of new nodes in a CBACO representation

4.2.2 Relations related to obligations

In the new relations, edges of type *OG* (resp. *DE*) will be green if *ge* (resp. *ev*) is *i* and red if it is *f*, others will be gray. As with edges of type *CC*, edges of type *GG* and *EE* will connect to the **In** port of a node if the *ent* value of the node is in the *target* value of the edge and to the **Out** port otherwise.

4.2.3 Color coding

The idea of color coding came as a way of distinguishing categories that belong to the permission/prohibition subgraph from the ones that belong to the obligation subgraph and the ones that are used in both. This color coding was then extended to every type of node in order to make the graph more informative. The coding is simple, if a node belongs solely to the permission/prohibition subgraph then its color is yellow, if it belongs solely to the obligation subgraph then its color is blue and if it belongs to both then its color is green. We use the color also to make explicit which of the event nodes has its *now* set to $\top$ by setting its color to a lighter blue. This lighter blue color is also used on duty nodes until event e_2 happens.

Example 4.2. In Figure 4.2 we have the graphical representation of a CBACO policy such that:

- $\mathcal{P} = \{J. Dorian, C. Tuck\};$
- $\mathcal{C} = \{Dr(J. Lewis), Dr(F. Mason)\};$
- $\mathcal{A} = \{Read, Declare\};$
- $\mathcal{R} = \{Rec(J. Lewis), Rec(F. Mason), Admin-log\};$
- $\subseteq = \emptyset;$
- $\subseteq_{\emptyset} = \emptyset;$
- $\mathcal{E} = \{\{act = Read, subj = C. Tuck, obj = Rec(J. Lewis), time = 120\}, \{act = Declare, subj = C. Tuck, obj = Admin-log, time = 200\}\};$
- $\mathcal{G} = \{gen_read[J. Dorian, F. Mason], gen_read[C. Tuck, J. Lewis]\};$
- $\mathcal{H} = \{\{\{act = Read, subj = C. Tuck, obj = Rec(J. Lewis), time = 120\}, \{act = Declare, subj = C. Tuck, obj = Admin-log, time = 200\}\}\};$

- $\mathcal{PCA} = \{(J. \text{Dorian}, \text{Dr}(J. \text{Lewis})), (C. \text{Tuck}, \text{Dr}(F. \text{Mason}))\};$
- $\mathcal{ARCA} = \{(Read, \text{Rec}(F. \text{Mason}), \text{Dr}(F. \text{Mason})), (Read, \text{Rec}(J. \text{Lewis}), \text{Dr}(J. \text{Lewis}))\};$
- $\mathcal{PAR} = \{(J. \text{Dorian}, Read, \text{Rec}(J. \text{Lewis})), (C. \text{Tuck}, Read, \text{Rec}(F. \text{Mason}))\};$
- $\mathcal{BARCA} = \emptyset;$
- $\mathcal{BAR} = \emptyset;$
- $\mathcal{UNDET} = \{(C. \text{Tuck}, Read, \text{Rec}(J. \text{Lewis})), (J. \text{Dorian}, \text{Declare}, \text{Rec}(J. \text{Lewis})),$
 $(C. \text{Tuck}, \text{Declare}, \text{Rec}(J. \text{Lewis})), (J. \text{Dorian}, Read, \text{Rec}(F. \text{Mason})),$
 $(J. \text{Dorian}, \text{Declare}, \text{Rec}(F. \text{Mason})), (C. \text{Tuck}, \text{Declare}, \text{Rec}(F. \text{Mason})),$
 $(J. \text{Dorian}, Read, \text{Admin-log}), (C. \text{Tuck}, Read, \text{Admin-log}),$
 $(J. \text{Dorian}, \text{Declare}, \text{Admin-log}), (C. \text{Tuck}, \text{Declare}, \text{Admin-log})\};$
- $\mathcal{OCA} = \{(\text{Declare}, \text{Admin-log}, \text{gen_read}[J. \text{Dorian}, F. \text{Mason}], \perp, \text{Dr}(J. \text{Lewis})),$
 $(\text{Declare}, \text{Admin-log}, \text{gen_read}[C. \text{Tuck}, J. \text{Lewis}], \perp, \text{Dr}(F. \text{Mason}))\};$
- $\mathcal{OPA} = \{(J. \text{Dorian}, \text{Declare}, \text{Admin-log}, \text{gen_read}[J. \text{Dorian}, F. \text{Mason}], \perp),$
 $(C. \text{Tuck}, \text{Declare}, \text{Admin-log}, \text{gen_read}[C. \text{Tuck}, J. \text{Lewis}], \perp)\};$
- $\mathcal{DA} = \{(C. \text{Tuck}, \text{Declare}, \text{Admin-log}, \{\text{act} = Read, \text{subj} = C. \text{Tuck},$
 $\text{obj} = \text{Rec}(J. \text{Lewis}), \text{time} = 120\}, \perp)\};$
- $\mathcal{ET} = \{(\{\text{act} = Read, \text{subj} = C. \text{Tuck}, \text{obj} = \text{Rec}(J. \text{Lewis})\},$
 $\text{gen_read}[C. \text{Tuck}, J. \text{Lewis}])\};$
- $\mathcal{EI} = \{(\{\text{act} = Read, \text{subj} = C. \text{Tuck}, \text{obj} = \text{Rec}(J. \text{Lewis}), \text{time} = 120\}, \{\text{act} =$
 $\text{Declare}, \text{subj} = C. \text{Tuck}, \text{obj} = \text{Admin-log}, \text{time} = 200\}, [\{\text{act} = Read, \text{subj} = C. \text{Tuck},$
 $\text{obj} = \text{Rec}(J. \text{Lewis}), \text{time} = 120\}, \{\text{act} = \text{Declare}, \text{subj} = C. \text{Tuck}, \text{obj} = \text{Admin-log},$
 $\text{time} = 200\}])\}.$

Note that the duty is not represented in the graph, this is due to the fact that we use rewrite rules and the strategy language to instantiate and update the state of duties in a history of events. As we can see the active event node (light blue circle) is connected to an event scheme node (blue ring), which is connected to an obligation node (blue hexagon) by a green edge ($ev = i$) meaning it will trigger a duty. This is shown in Figure 4.3a. The only changes in the graph is the addition of this duty node (light blue hexagon) and respective edges and the change of the active event (light blue circle).

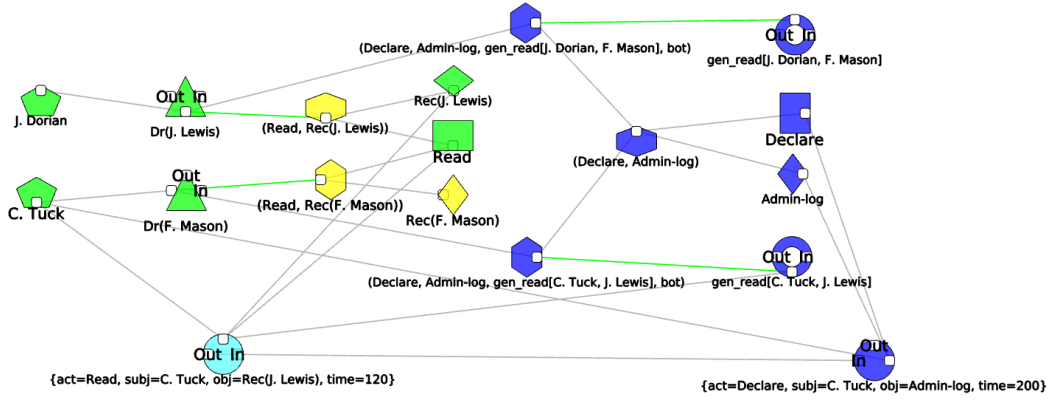
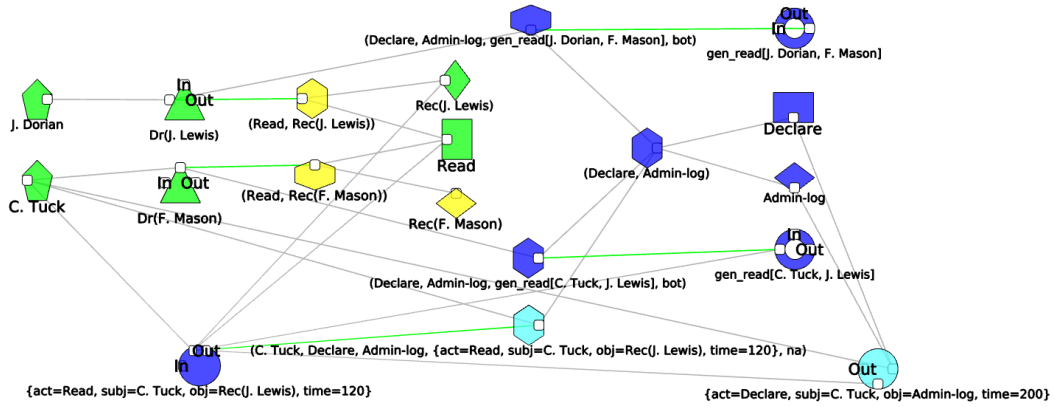
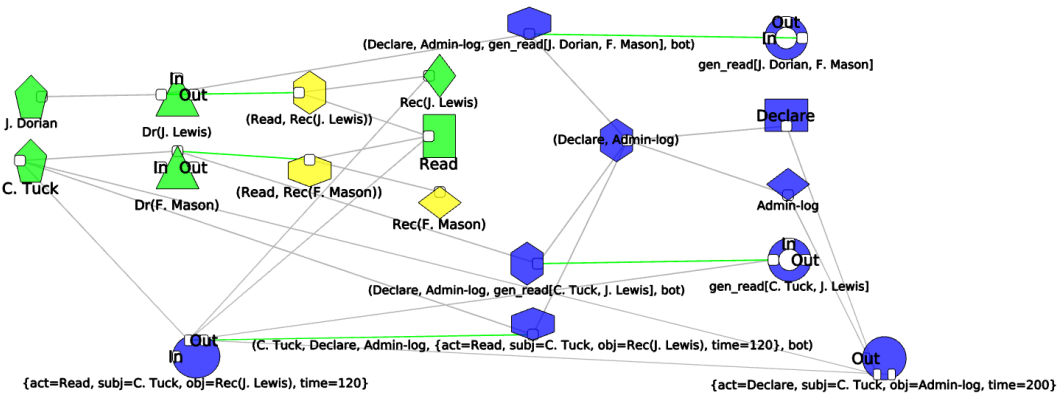


FIGURE 4.2: CBACO policy representation

Finally, the active event in Figure 4.3a represents the action on the resource by the subject needed to fulfill the duty. The final graph is shown in Figure 4.3b with the only changes being there is no active event (light blue circle) as the history has ended and the duty node is now blue and its viewLabel (and ent) matches the duty in the policy.



(A) After first event is processed



(B) After both events are processed

FIGURE 4.3: CBACO policy representations

Now that we have seen the implementation of the static part of our framework in PORGY. We will describe, in the next chapter, how we implemented the dynamic part associated with the CBACO model using the strategies and rewrite rules of PORGY.

Chapter 5

Dynamic CBAC(O) in PORGY

In this chapter we give examples of some of the rewrite rules and strategies we created in the PORGY tool to implement the dynamic part of our framework. They are presented in the same order that we discussed them in Section 3.3. While in the section covering the duties related rules and strategies we show every rule and strategy needed, in the others we cover only a few examples. In the scenario-specific ones in Section 5.1 this is due to the fact that these rules will depend on the specific policy being represented, in the remaining sections this is because of the big number of rules and the fact that they are similar to each other. Nonetheless, we mention all the rules we created and, through the examples, it should be possible to extrapolate the creation of all the remaining rules. As in the previous chapter, all the nodes and edges have only the necessary attributes and, depending on the specific policy being represented, one could add more.

5.1 PCA

A key point of CBAC(O) policies is that the assignment of principals, permissions, prohibitions and obligations to categories is not static, it depends on the system state. We will now focus on an example where the principal-category assignment depends on the properties of principals and see how we can use rewrite rules and the strategy language to ensure that relations on the graph are correct.

Example 5.1. *Consider a CBAC policy where:*

- $\mathcal{P} = \{P. Flowers, C. Espinosa, L. Roberts\};$
- $\mathcal{C} = \{Registered Nurse, Nurse Practitioner\};$

- $\mathcal{A} = \{\text{Cancel}, \text{Perform}\};$
- $\mathcal{R} = \{\text{Lab order}, \text{Specimen collection}\};$
- $\subseteq = \{(\text{Registered Nurse}, \text{Nurse Practitioner})\};$
- $\mathcal{ARCA} = \{(\text{Cancel}, \text{Lab order}, \text{Registered Nurse});$
 $(\text{Perform}, \text{Specimen collection}, \text{Nurse Practitioner})\};$

and the $\mathcal{PCA}$ relation is dynamic and, by Axiom 2.1, so is $\mathcal{PAR}$. The $\mathcal{PCA}$ relation will depend on the qualification of each principal, therefore, in the record of each one, there will be an attribute qualification which value specifies the category it belongs to. The starting graph is shown in Figure 5.1 where there are no edges of type $\mathcal{PC}$ and the value of attribute qualification is Registered Nurse for principal L. Roberts and Nurse Practitioner for principals P. Flowers and C. Espinosa.

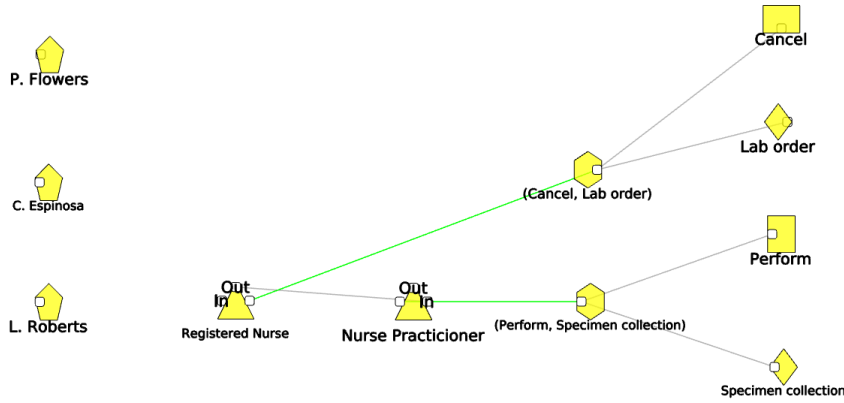


FIGURE 5.1: Starting graph

To make the $\mathcal{PC}$ edges we use two rules, one for each category, with names *conRegisteredNurse* and *conNursePractitioner*. In Figure 5.2 we see rule *conRegisteredNurse* where:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2, p_3, p_4\}$, $E_L = \emptyset$, $W = \{n_1\}$ and:
 - $\mathcal{L}_L(n_1) = \{(\text{Interface}, \{p_1\}), (\text{ent}, v_1), (\text{type}, P), (\text{viewLabel}, v_2),$
 $(\text{qualification}, \text{Registered Nurse})\};$
 - $\mathcal{L}_L(n_2) = \{(\text{Interface}, \{p_2, p_3, p_4\}), (\text{ent}, \text{Registered Nurse}), (\text{type}, C),$
 $(\text{viewLabel}, \text{Registered Nurse})\};$
 - $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_L(p_i) = \{(\text{Attach}, n_i), (\text{Arity}, 0)\};$
 - $\mathcal{L}_L(p_3) = \{(\text{Attach}, n_2), (\text{Arity}, 0), (\text{viewLabel}, \text{In})\};$
 - $\mathcal{L}_L(p_4) = \{(\text{Attach}, n_2), (\text{Arity}, 0), (\text{viewLabel}, \text{Out})\}.$

- R_M^N is defined by $N_R = \{n_1, n_2\}$, $P_R = \{p_1, p_2, p_3, p_4\}$, $E_R = \{e_1\}$, $M = \{n_1\}$, $N = \emptyset$ and:

- $\forall n_i \in N_R, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
- $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_R(p_i) = \{(Attach, n_i), (Arity, 1)\}$;
- $\forall p_i \in \{p_3, p_4\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i)$;
- $\mathcal{L}_R(e_1) = \{(Connect, \{p_1, p_2\}), (type, PC)\}$.

Rule *conNursePractitioner* is obtained from this one by replacing every occurrence of "Registered Nurse" with "Nurse Practitioner".

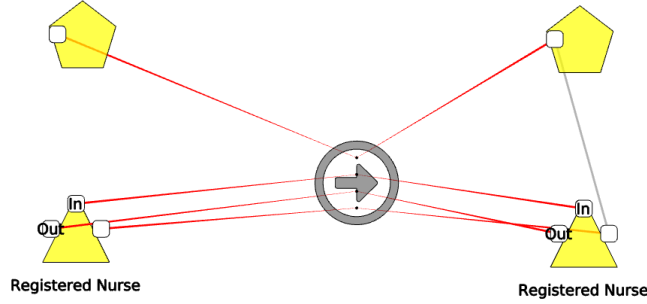


FIGURE 5.2: Rewrite rule *conRegisteredNurse*

With these two rules we could make a strategy that connects principals to the right category, which would work for the graph in Figure 5.1. But our aim is to have a strategy that not only makes the right connections but also updates them if the qualification of any principal changes. For this purpose we also have rule *rmPC*, which will delete edges of type PC. This rule is shown in Figure 5.3 and:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2, p_3, p_4\}$, $E_L = \{e_1\}$, $W = \{n_1\}$ and:
- $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (qualification, v_3)\}$;
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_2, p_3, p_4\}), (ent, v_4), (type, C), (viewLabel, v_5)\}$;
 - $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\}$;
 - $\mathcal{L}_L(p_3) = \{(Attach, n_2), (Arity, 0), (viewLabel, In)\}$;
 - $\mathcal{L}_L(p_4) = \{(Attach, n_2), (Arity, 0), (viewLabel, Out)\}$;

- $\mathcal{L}_L(e_1) = \{(Connect, \{p_1, p_2\}), (type, PC)\}$.
- R_M^N is defined by $N_R = \{n_1, n_2\}$, $P_R = \{p_1, p_2, p_3, p_4\}$, $E_R = \emptyset$, $M = \{n_1\}$, $N = \emptyset$ and:
 - $\forall n_i \in N_R, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
 - $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_R(p_i) = \{(Attach, n_i), (Arity, 0)\}$;
 - $\mathcal{L}_R(p_3) = \{(Attach, n_2), (Arity, 0), (viewLabel, In)\}$;
 - $\mathcal{L}_R(p_4) = \{(Attach, n_2), (Arity, 0), (viewLabel, Out)\}$.

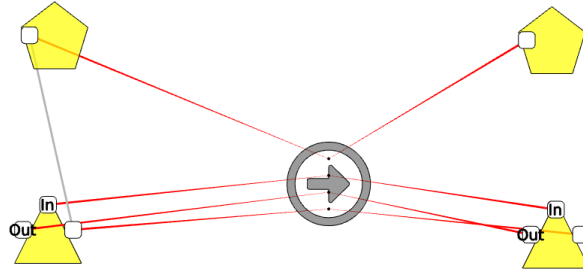


FIGURE 5.3: Rewrite rule rmPC

With this three rewrite rules we write the following strategy named *pca*:

```

setPos(all(property(crtGraph,node,type=="P")));
repeat(one(rmPC));
setBan(all(crtPos));
while(not(isEmpty(crtBan)))do(
  setPos(one(crtBan));
  setBan(all(crtBan\crtPos));
  try(one(conRegisteredNurse));
  try(one(conNursePractitioner))
)

```

Applying this strategy to the graph in Figure 5.1 we get the graph in Figure 5.4. If we then change the qualification of principal C. Espinosa to Registered Nurse and apply the strategy again we get the graph in Figure 5.5.

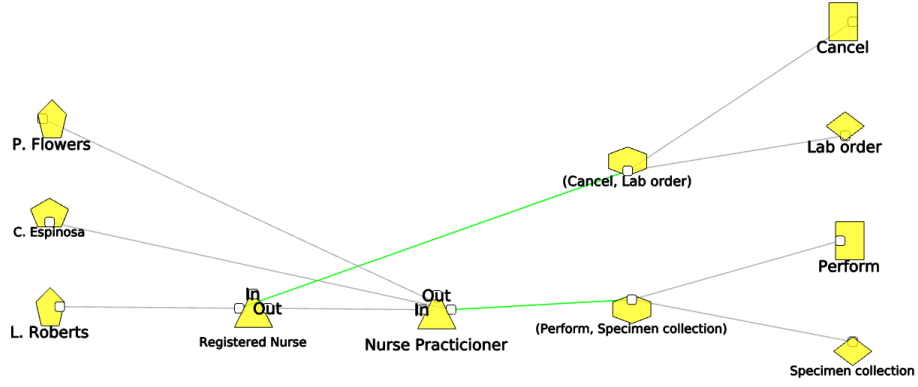


FIGURE 5.4: Result of applying pca to starting graph

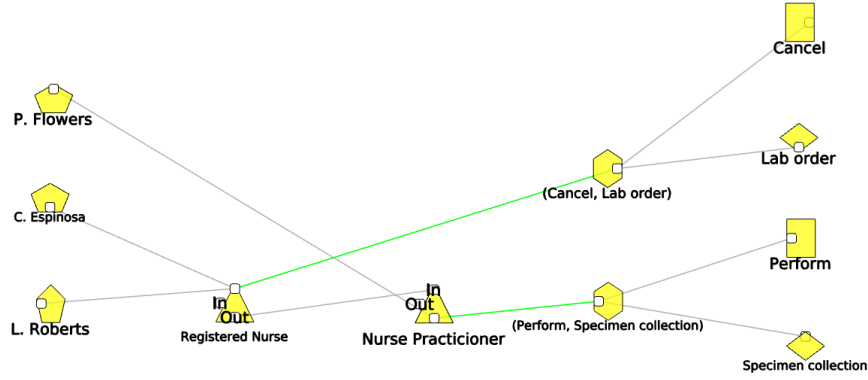


FIGURE 5.5: Result of applying pca after changing qualification of C. Espinosa

5.2 Auxiliary edges

In this section we talk about the creation (and deletion) of auxiliary edges. These edges are defined as redundant or non existent by the model but are needed so we can create the general rewrite rules presented in the next sections. Their purpose is to make implicit relations explicit, nonetheless, they will be invisible so they can be used in rewrite rules but will not appear in the graph visualization. They are as follows:

- auxiliary edges of type PC which represent that $\exists c, c' \in \mathcal{C}, (p, c) \in \mathcal{PCA} \wedge c \subseteq c'$ (resp. $\exists c, c' \in \mathcal{C}, (p, c) \in \mathcal{PCA} \wedge c \subseteq_{\mathcal{O}} c'$) in axiom 2.1 (resp. 2.5), which is represented in the graph as paths of type $PC, (\vec{CC}_R)^*$ (resp. $PC, (\vec{CC}_O)^*$) with length greater or equal to 2. To distinguish them we will use the attributes *auth* and *obl* as in edges of type CC ;

- auxiliary edges of type CP_r which represent that $\exists c, c' \in \mathcal{C}, c' \subseteq c \wedge (a, r, c') \in \mathcal{BARCA}$ in axiom 2.2 which is represented in the graph as paths of type $\overleftarrow{CC}_{Pr}, CP_r$ with length greater or equal to 2;
- auxiliary edges of type PP_r which represent $\mathcal{PAR}$ and $\mathcal{BAR}$ relations represented in the graph as paths of type $PC, (\overrightarrow{CC}_{Pr})^*, CP_r^A$ and $PC, (\overleftarrow{CC}_{Pr})^*, CP_r^B$, respectively. To distinguish them we will use the attribute *auth* as in edges of type CP_r ;
- auxiliary edges of type PO which represent $\mathcal{OPA}$ relations represented in the graph as paths of type $PC, (\overrightarrow{CC}_O)^*, CO$;
- auxiliary edges of type EG which represent $\mathcal{ET}$ relations represented in the graph as paths of type $EG, (\overrightarrow{GG})^*$ with length greater or equal to 2.

All of them will have attribute *aux* set to *true*. An example of a rewrite rule to create an auxiliary edge of type PC was already shown in Figure 2.3, where the edge was made visible for demonstrative purposes. Furthermore the strategy used to create them is close to the one in Example 2.9.

Example 5.2. To demonstrate the process of making the rules and strategies needed to generate these edges we will focus on the creation of ones that represent paths of type $PC, (\overrightarrow{CC}_{Pr})^*$, we will refer to them as of type PC_{Pr} . The process of making rules and strategies for the remaining edges is parallel to this. As the edges we will create are invisible they will be highlighted in blue when showing rewrite rules.

The rules will be composed of:

- 3 nodes (1 of type P and 2 of type C) in the left side and in the right side;
- 2 edges (1 of type PC and 1 of type CC_{Pr}) in the left side and 3 edges (1 of type PC , 1 of type PC_{Pr} and 1 of type CC_{Pr}) in the right side.

In Figure 5.6 we show the first of these rules where:

- L_W is defined by $N_L = \{n_1, n_2, n_3\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, $E_L = \{e_1, e_2\}$, $W = \{n_1, n_2\}$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, yellow)\}$;

- $\mathcal{L}_L(n_2) = \{(Interface, \{p_2, p_3, p_4\}), (ent, v_3), (type, C), (viewLabel, v_4), (viewColor, yellow)\};$
 - $\mathcal{L}_L(n_3) = \{(Interface, \{p_5, p_6, p_7\}), (ent, v_5), (type, C), (viewLabel, v_6), (viewColor, yellow)\};$
 - $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\};$
 - $\mathcal{L}_L(p_3) = \{(Attach, n_2), (Arity, 1), (viewLabel, v_7)\};$
 - $\mathcal{L}_L(p_4) = \{(Attach, n_2), (Arity, 0), (viewLabel, v_8)\};$
 - $\mathcal{L}_L(p_5) = \{(Attach, n_3), (Arity, 0)\};$
 - $\mathcal{L}_L(p_6) = \{(Attach, n_3), (Arity, 1), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_7) = \{(Attach, n_3), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_1, p_2\}), (type, PC), (aux, false)\};$
 - $\mathcal{L}_L(e_2) = \{(Connect, \{p_3, p_6\}), (type, CC), (auth, \top), (obl, v_9), (target, v_{10})\}.$
- R_M^N is defined by $N_R = \{n_1, n_2, n_3\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, $E_R = \{e_1, e_2, e_3\}$, $M = \{n_1, n_2\}$, $N = \{n_3\}$ and:
- $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i);$
 - $\forall p_i \in P_L \setminus \{p_1, p_5\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i);$
 - $\mathcal{L}_R(p_1) = \{(Attach, n_1), (Arity, 2)\};$
 - $\mathcal{L}_L(p_5) = \{(Attach, n_3), (Arity, 1)\};$
 - $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i);$
 - $\mathcal{L}_R(e_3) = \{(Connect, \{p_1, p_5\}), (type, PC), (aux, true), (auth, \top), (viewColor, invisible)\}.$

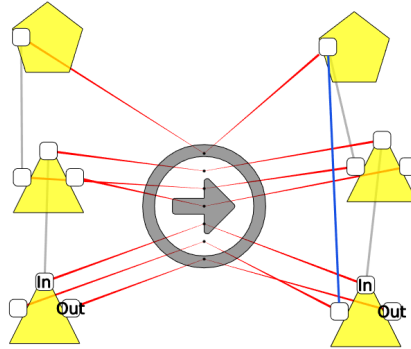


FIGURE 5.6: Rewrite rule auxPCpyyy

As you can see only one of the nodes of type C has its ports with their respective names, this is because we only need to ensure that the edge of type CC_{Pr} connects to the **In** port of that node.

As we will need to create edges of type PC_{Pr} from other edges of type PC_{Pr} and there is no way of preserving the color of edges present in rewrite rules (using a value v_i such as we use to keep other attributes from L to R), from this rule we will create another which is shown in Figure 5.7a. The only difference from the previous rule to this one is:

- $\mathcal{L}_L(e_1) = \{(\text{Connect}, \{p_1, p_2\}), (\text{type}, PC), (\text{aux}, \text{true}), (\text{auth}, \top), (\text{viewColor}, \text{invisible})\}.$

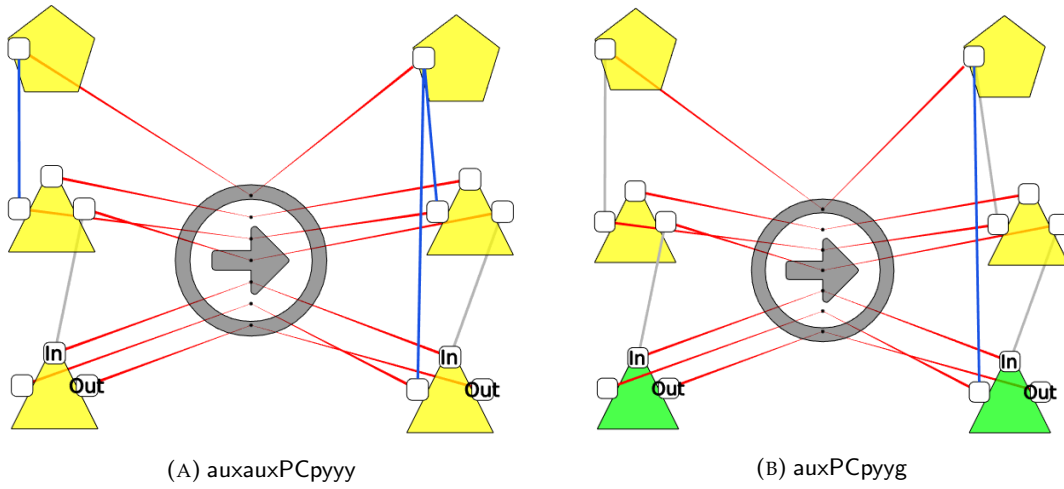


FIGURE 5.7: Rewrite rules

As with edges, the color of nodes present in the rewrite rules cannot be preserved, so we will have to create a version of these two rules for every possible arrangement of node colors. We know these nodes belong to the permission/prohibition subgraph so each of them can be either yellow or green, so from the two rules we have created so far we will create fourteen more, one of which is shown in Figure 5.7b. The difference between this and the first rule is:

- $\mathcal{L}_L(n_3) = \{(\text{Interface}, \{p_5, p_6, p_7\}), (\text{ent}, v_5), (\text{type}, C), (\text{viewLabel}, v_6), (\text{viewColor}, \text{green})\}.$

To use these rules we use three strategies, two just to combine all the different rewrite rules using the *orelse* construct (one for the *auxPC* rules and one for the *auxauxPC* ones) and the other to actually control the way they are applied. One of the first two, called *auxPCrr* (the suffix *rr* stands for rewrite rules and means this is just an agglomeration of all the rewrite rules with the same purpose) is as follows:


```

(((one(auxPCpyyy))orelse(one(auxPCpyyg))
  )orelse((one(auxPCpygy))orelse(one(auxPCpygg)))
)orelse(((one(auxPCpgyy))orelse(one(auxPCpgyg))
  )orelse((one(auxPCpggy))orelse(one(auxPCpggg))))

```

Before presenting the other strategy it is worth noting that in the rules created the top two nodes of L belong to W , the top two nodes of R belong to M and the bottom one to N . By adding the node to which was made the new connection to the banned subgraph we accomplish two things:

1. make sure that we do not create more than one edge connecting the same pair principal/category;
2. keep track of the new categories to which a connection was made.

With this in mind the remaining strategy called *auxPCp* is:

```

setBan(all(property(crtGraph,node,type=="P")));
while(not(isEmpty(crtBan)))do(
  setPos(one(crtBan));
  setBan(all(crtBan\crtPos));
  setPos(all(crtPos[cup]ngb(crtPos,edge,type=="PC")));
  if(#auxPCpr#)then(
    repeat(#auxPCpr#);
    setPos(all(property(crtPos,node,type=="P")[cup]
      property(crtBan,node,type=="C")));
    setBan(all(property(crtBan,node,type=="P")));
    while(#auxauxPCpr#)do(
      repeat(#auxauxPCpr#);
      setPos(all(property(crtPos,node,type=="P")[cup]
        property(crtBan,node,type=="C")));
      setBan(all(property(crtBan,node,type=="P")))
    )
  )
)

```

To delete the edges we will need to create rewrite rules as well, this rules will be composed of:

- two nodes (one of type P and one of type C) in the left side and in the right side;
- one edge of type PC_P in the left side.

Because each of the nodes can be either yellow or green there will be four such rules, one of which is shown in Figure 5.8, where:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2, p_3, p_4\}$, $E_L = \{e_1\}$, $W = \emptyset$ and:
 - $\mathcal{L}_L(n_1) = \{(\text{Interface}, \{p_1\}), (\text{ent}, v_1), (\text{type}, P), (\text{viewLabel}, v_2), (\text{viewColor}, \text{yellow})\}$;
 - $\mathcal{L}_L(n_2) = \{(\text{Interface}, \{p_2, p_3, p_4\}), (\text{ent}, v_3), (\text{type}, C), (\text{viewLabel}, v_4), (\text{viewColor}, \text{yellow})\}$;
 - $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_L(p_i) = \{(\text{Attach}, n_i), (\text{Arity}, 1)\}$;
 - $\mathcal{L}_L(p_3) = \{(\text{Attach}, n_3), (\text{Arity}, 0), (\text{viewLabel}, \text{In})\}$;
 - $\mathcal{L}_L(p_4) = \{(\text{Attach}, n_3), (\text{Arity}, 0), (\text{viewLabel}, \text{Out})\}$;
 - $\mathcal{L}_L(e_1) = \{(\text{Connect}, \{p_1, p_2\}), (\text{aux}, \text{true}), (\text{auth}, \top)\}$.
- R_M^N is defined by $N_R = \{n_1, n_2\}$, $P_R = \{p_1, p_2, p_3, p_4\}$, $E_R = \emptyset$, $M = \emptyset$, $N = \emptyset$ and:
 - $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
 - $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_R(p_i) = \{(\text{Attach}, n_i), (\text{Arity}, 0)\}$;
 - $\forall p_i \in \{p_3, p_4\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i)$.

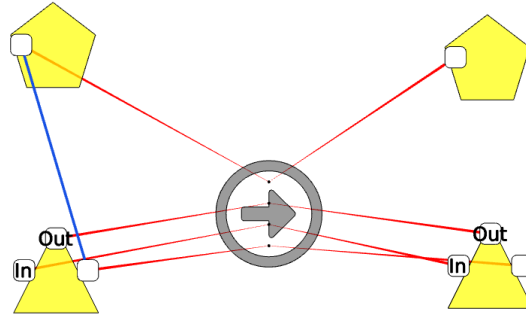


FIGURE 5.8: Rewrite rule rmauxPCpyy

To use these rewrite rules we use two strategies, one to combine the different rewrite rules using the *orElse* construct and the other to actually control their application. The first one, called *rmauxPCpr*, is:

```
((one(rmauxPCpyy))orElse(one(rmauxPCpyg)))
)orElse((one(rmauxPCpgy))orElse(one(rmauxPCpgg)))
```

The second one, called *rmauxPCp*, is simply:

```
repeat(#rmauxPCprr#)
```

5.2.1 Auxiliary nodes and edges for $\mathcal{UNDET}$

We talk about the creation of auxiliary nodes and edges for $\mathcal{UNDET}$ separately because, although they are auxiliary as the previously discussed edges, they will only be used for calculating the $\mathcal{UNDET}$ relation and for this reason they will be created as visible. Furthermore, the calculation of $\mathcal{UNDET}$ is different because we will have to create nodes that are not in the graph.

Example 5.3. Consider a CBAC policy where:

- $\mathcal{P} = \{P. Flowers, C. Espinosa\};$
- $\mathcal{C} = \{Registered Nurse, Nurse Practitioner\};$
- $\mathcal{A} = \{Create, Perform\};$
- $\mathcal{R} = \{Prescription, Specimen collection\};$
- $\subseteq = \{(Registered Nurse, Nurse Practitioner)\};$
- $ARCA = \{(Perform, Specimen collection, Nurse Practitioner)\};$
- $\mathcal{PAR} = \{(P. Flowers, Perform, Specimen collection),$
 $(C. Espinosa, Perform, Specimen collection)\};$
- $BARCA = \{(Create, Prescription, Registered Nurse)\};$
- $BAR = \{(P. Flowers, Create, Prescription), (C. Espinosa, Create, Prescription)\};$
- $\mathcal{UNDET} = \{(P. Flowers, Create, Specimen collection),$
 $(C. Espinosa, Create, Specimen collection), (P. Flowers, Perform, Prescription),$
 $(C. Espinosa, Perform, Prescription)\}.$

The representation of this policy is shown in Figure 5.9.

We begin by demonstrating how to create the new nodes. To represent the $\mathcal{UNDET}$ relation we need to have a node of type P_r for each pair of nodes with type A and R . As so the rewrite rules to create these nodes will be composed of two nodes (one of type A and one of type R) in the left side and three nodes (one of type A , one of type R and one of type P_r) in the right side.

One of these rules is shown in Figure 5.10, where:

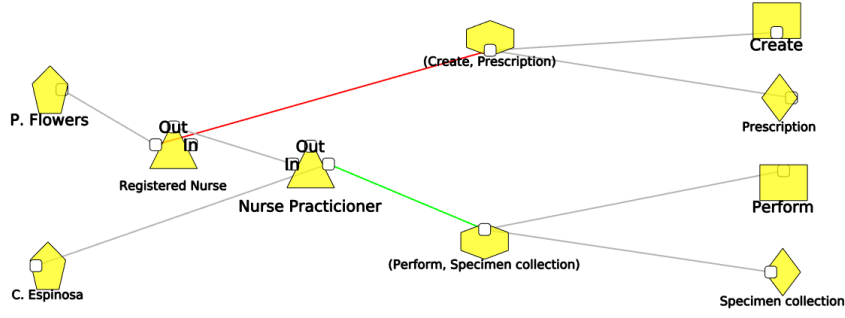


FIGURE 5.9: CBAC policy representation

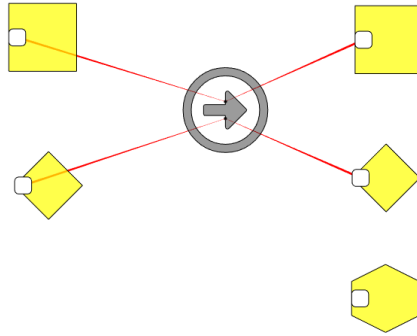


FIGURE 5.10: Rewrite rule auxPryy

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2\}$, $E_L = \emptyset$, $W = \{n_1\}$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, A), (viewLabel, v_2), (viewColor, yellow)\}$;
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, R), (viewLabel, v_4), (viewColor, yellow)\}$;
 - $\forall p_i \in P_L, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 0)\}$;
- R_M^N is defined by $N_R = \{n_1, n_2, n_3\}$, $P_R = \{p_1, p_2, p_3\}$, $E_R = \emptyset$, $M = \{n_1\}$, $N = \{n_2\}$ and:
 - $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
 - $\mathcal{L}_R(n_3) = \{(Interface, \{p_3\}), (ent, (v_1, v_3)), (type, P_r), (aux, true), (viewLabel, (v_2, v_4)), (viewColor, yellow)\}$;
 - $\forall p_i \in P_R, \mathcal{L}_R(p_i) = \{(Attach, n_i), (Arity, 0)\}$.

As the color of nodes is not preserved by rewrite rules we will create one version of this rule for each possible combination of the colors that nodes of type A and R can have. Because they both can be yellow, green or blue we will have to create nine versions of this rule. To use these rules we use two strategies, one just to combine all rules using the *orelse* construct and the other to actually control the application of these rules. The first one, called *auxPrrr*, is as follows:

```
((((one(auxPryy))orelse(one(auxPryg)))orelse(one(auxPryb))
  )orelse((one(auxPrgy))orelse(one(auxPrgg)))
)orelse(((one(auxPrbg))orelse(one(auxPrby))
  )orelse((one(auxPrbg))orelse(one(auxPrbb)))))
```

Before presenting the second strategy it is worth noting that, in the rules, in L the node of type A belongs to W and in R the node of type A is in M and the node of type R is in N . As before we put the node of type R in the banned subgraph to avoid creating the same node of type P_r twice. The second strategy, called *auxPr*, is:

```
setBan(all(property(crtGraph,node,type=="A")));
while(not(isEmpty(crtBan)))do(
  setPos(one(crtBan));
  setBan(all(crtBan\crtPos));
  setBan(all(crtBan[cup]
    ngb(ngb(crtPos,edge,type=="PrA"),edge,type=="PrR"))));
  repeat(#auxPrrr#);
  setBan(all(property(crtBan,node,type=="A")))
)
```

The rewrite rules to create the edges will be composed of:

- two nodes (one of type P and one of type P_r) in both sides;
- one edge of type PP_r in the right side.

One of these rules is shown in Figure 5.11, where:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2\}$, $E_L = \emptyset$, $W = \{n_2\}$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, yellow)\}$;

- $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, P_r), (viewLabel, v_4), (viewColor, yellow)\};$
- $\forall p_i \in P_L, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 0)\};$
- R_M^N is defined by $N_R = \{n_1, n_2\}, P_R = \{p_1, p_2\}, E_R = \{e_1\}, M = \{n_2\}, N = \{n_1\}$ and:
 - $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i);$
 - $\forall p_i \in P_R, \mathcal{L}_R(p_i) = \{(Attach, n_i), (Arity, 1)\};$
 - $\mathcal{L}_R(e_1) = \{(Connect, \{p_1, p_2\}), (type, PP_r), (auth, U), (aux, true)\}.$

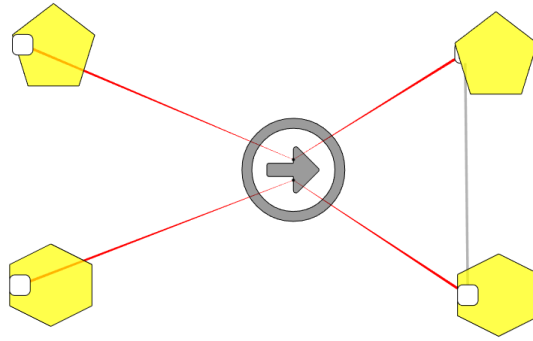


FIGURE 5.11: Rewrite rule auxPPruyy

Because either of the nodes can be yellow, green or blue we will create nine versions of this rule for each possible combination of colors. Furthermore, we will need a slightly different set of rules to create edges connecting to the newly created nodes of type P_r . The only difference from the previous rules to these ones is:

- $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, P_r), (aux, true), (viewLabel, v_4), (viewColor, yellow)\}.$

Because the created nodes are all yellow we will only have three of these rules, their names are *auxauxPPruyy*, *auxauxPPrugy* and *auxauxPPruby*. The strategies that combine these rules using the *orelse* construct, called *auxPPrurr* and *auxauxPPrurr*, respectively, are as follows:

```

(((one(auxPPruyy))orelse(one(auxPPrugy)))orelse(one(auxPPruby))
  )orelse((one(auxPPrugy))orelse(one(auxPPrugg)))
)orelse(((one(auxPPrugb))orelse(one(auxPPruby))
  )orelse((one(auxPPrubg))orelse(one(auxPPrubbb)))))

```

`(one(auxauxPPruy))orelse((one(auxauxPPrugy))orelse(one(auxauxPPruby)))`

Before showing the full strategy to create the nodes and edges of the $UNDET$ relation there is still a set of rules we need to discuss. In the process of creating the edges of type PP_r^U we will create edges of type PP_r^A and PP_r^B . These later edges will have to be deleted afterwards, for this purpose we create the rewrite rule $r_{auxPPrnuyy}$ (and 8 others for each possible combination of colors the nodes can have) shown in Figure 5.12, where:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2\}$, $E_L = \{e_1\}$, $W = \emptyset$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, yellow)\}$;
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, P_r), (viewLabel, v_4), (viewColor, yellow)\}$;
 - $\forall p_i \in P_L, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\}$;
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_1, p_2\}), (type, PP_r), (aux, true), (auth, \neq U)\}$.
- R_M^N is defined by $N_R = \{n_1, n_2\}$, $P_R = \{p_1, p_2\}$, $E_R = \emptyset$, $M = \emptyset$, $N = \emptyset$ and:
 - $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
 - $\forall p_i \in P_R, \mathcal{L}_R(p_i) = \{(Attach, n_i), (Arity, 0)\}$;

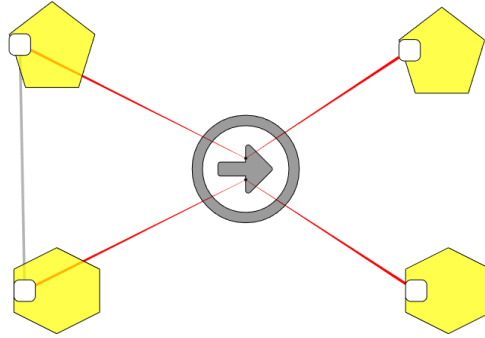


FIGURE 5.12: Rewrite rule $r_{auxPPrnuyy}$

As you can see the $auth$ value of the edge is set to be different from U so that the rule will apply to edges with $auth$ set to either A or B . The strategies used to apply these rules, called $r_{auxPPrnu}$ and $r_{auxPPrnuur}$, respectively, are as follows:

`repeat(#rmauxPPrnuur#)`

```

((((one(rmauxPPrnuyy))orelse(one(rmauxPPrnuyg))
  )orelse((one(rmauxPPrnugg))orelse(one(rmauxPPrnugy))))
)orelse((((one(rmauxPPrnuby))orelse(one(rmauxPPrnuyb))
  )orelse((one(rmauxPPrnugb))orelse(one(rmauxPPrnubg))))
)orelse(one(rmauxPPrnubb))

```

Having done all this the strategy to calculate the $UNDET$ relation, called $auxPPru$ is:

```

#auxPCp#;#auxCPr#;
setBan(all(property(crtGraph,node,type=="Pr")));
while(not(isEmpty(crtBan)))do(
  setPos(one(crtBan));
  setBan(all(crtBan\crtPos));
  repeat(#auxPPrarr#);repeat(#auxPPrbrr#);repeat(#auxPPrurr#);
  setBan(all(property(crtBan,node,type=="Pr")))
);
#rmauxPPrnu#;#rmauxPCp#;#rmauxCPr#;#auxPr#;
setBan(all(property(crtGraph,node,type=="Pr" && aux=="true")));
while(not(isEmpty(crtBan)))do(
  setPos(one(crtBan));
  setBan(all(crtBan\crtPos));
  repeat(#auxauxPPrurr#);
  setBan(all(property(crtBan,node,type=="Pr")))
)

```

Where $auxCPr$ and $rmauxCPr$ are the strategies used to create and remove the auxiliary edges of type CP_r needed by the rewrite rules of strategy $auxPPrbrr$ to create auxiliary edges of type PP_r^B . Similarly, the rewrite rules of strategy $auxPPrarr$ that creates the edges of type PP_r^A need the edges created by the $auxPCp$ strategy that we have already seen.

The strategy works by:

1. creating the edges of type PP_r^U for every node of type P_r already in the graph (first while cycle);
2. creating the auxiliary nodes of type P_r ($auxPr$ strategy);
3. creating the edges of type PP_r^U for the auxiliary nodes of type P_r (second while cycle).

In the first while cycle we create the edges of type PP_r^U simply by applying the *auxPPrurr* strategy after *auxPPrarr* and *auxPPrbrr* because, as in the rules used in *auxPPrurr*, the rules used by these two strategies add the nodes of type P to which connections are made to the banned subgraph. Therefore, after they are applied the only nodes of type P not in the banned subgraph will be those to which the node of type P_r has not been connected yet (not in $\mathcal{PAR}$ or $\mathcal{BAR}$ relation).

In the second cycle, as the new nodes of type P_r were not even in the graph (have no $\mathcal{ARCA}$ or $\mathcal{BARCA}$ relations and, therefore, no $\mathcal{PAR}$ or $\mathcal{BAR}$ relations), we just connect every auxiliary node of type P_r with every node of type P .

Applying this strategy to the graph in Figure 5.9 we get the graph shown in Figure 5.13.

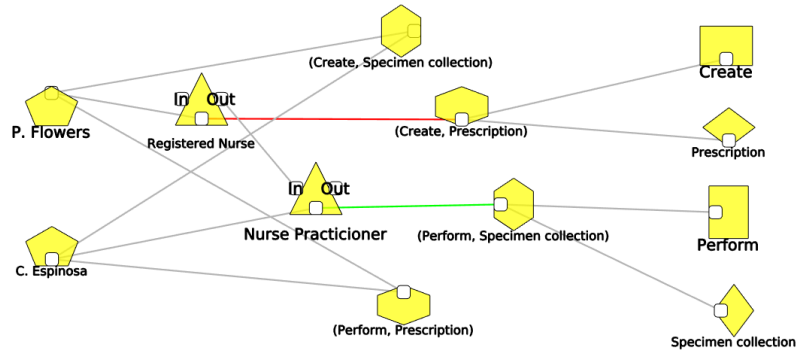


FIGURE 5.13: Result of applying strategy *auxPPru*

5.3 Duties

An example of the calculation of duties has already been shown in Example 4.2. For this purpose we will need to create rewrite rules and strategies to iterate through events, instantiate duties and update their state. It should be noted that the way we implement this part of the model is just one approach to it and others could be considered. In fact it is possible for the duties calculated by our implementation not to cover all the duties given by Axiom 2.6. Consider for example an obligation $o = (a, r, ge_1, ge_2)$ that is assigned to some principal p such that there is a history h with an event e_1 of type ge_1 and two events e_2, e_3 of type ge_2 that occur after e_1 . By Axiom 2.6 both duties (p, a, r, e_1, e_2) and (p, a, r, e_1, e_3) would be considered, but by our implementation only the one ending first will appear.

Example 5.4. Lets begin with the rewrite rules and strategies for the events part of the calculation. The rule to iterate through events, shown in Figure 5.14, is composed of two nodes and one edge in both sides, where:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $E_L = \{e_1\}$, $W = \{n_1\}$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1, p_2, p_3\}), (ent, v_1), (type, E), (viewLabel, v_2), (now, \top)\};$
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_4, p_5, p_6\}), (ent, v_3), (type, E), (viewLabel, v_4)\};$
 - $\mathcal{L}_L(p_1) = \{(Attach, n_1), (Arity, 0)\};$
 - $\mathcal{L}_L(p_2) = \{(Attach, n_1), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_3) = \{(Attach, n_1), (Arity, 1), (viewLabel, Out)\};$
 - $\mathcal{L}_L(p_4) = \{(Attach, n_2), (Arity, 0)\};$
 - $\mathcal{L}_L(p_5) = \{(Attach, n_2), (Arity, 1), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_6) = \{(Attach, n_2), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_3, p_5\}), (type, EE), (target, v_5)\}.$
- R_M^N is defined by $N_R = \{n_1, n_2\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $E_R = \{e_1\}$, $M = \emptyset$, $N = \emptyset$ and:
 - $\mathcal{L}_R(n_1) = \{(Interface, \{p_1, p_2, p_3\}), (ent, v_1), (type, E), (viewLabel, v_2), (now, \perp), (viewColor, blue)\};$
 - $\mathcal{L}_R(n_2) = \{(Interface, \{p_4, p_5, p_6\}), (ent, v_3), (type, E), (viewLabel, v_4), (now, \top), (viewColor, light\ blue)\};$
 - $\forall p_i \in P_L, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i);$
 - $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i).$

To use it we have strategy *nextE*:

```
setPos(all(property(crtGraph,node,now=="T")));
one(nextE)
```

When the iteration is over we have the rewrite rule shown in Figure 5.15, composed of one node in each side, where:

- L_W is defined by $N_L = \{n_1\}$, $P_L = \{p_1, p_2, p_3\}$, $E_L = \emptyset$, $W = \{n_1\}$ and:

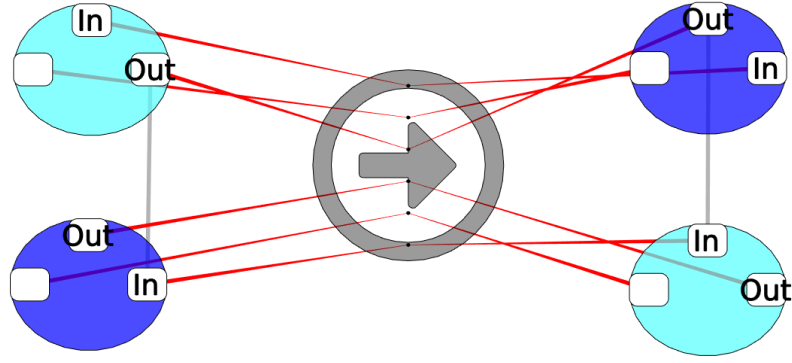


FIGURE 5.14: Rewrite rule nextE

- $\mathcal{L}_L(n_1) = \{(Interface, \{p_1, p_2, p_3\}), (ent, v_1), (type, E), (viewLabel, v_2), (now, \top)\};$
- $\mathcal{L}_L(p_1) = \{(Attach, n_1), (Arity, 0)\};$
- $\mathcal{L}_L(p_2) = \{(Attach, n_1), (Arity, 0), (viewLabel, In)\};$
- $\mathcal{L}_L(p_3) = \{(Attach, n_1), (Arity, 0), (viewLabel, Out)\}.$
- R_M^N is defined by $N_R = \{n_1\}, P_R = \{p_1, p_2, p_3\}, E_R = \emptyset, M = \emptyset, N = \emptyset$ and:
 - $\mathcal{L}_R(n_1) = \{(Interface, \{p_1, p_2, p_3\}), (ent, v_1), (type, E), (viewLabel, v_2), (now, \perp), (viewColor, blue)\};$
 - $\forall p_i \in P_L, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i).$

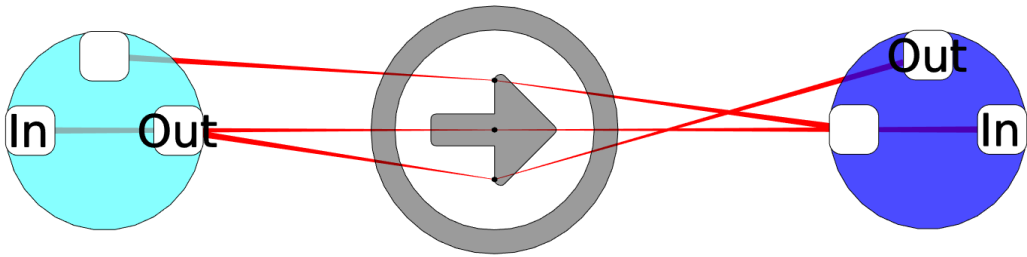


FIGURE 5.15: Rewrite rule closeH

To use it we have strategy *closeH*:

```
setPos(all(property(crtGraph,node,now=="T")));
one(closeH)
```

Lastly, regarding events, we have two strategies just to check if the event with now set to $\top$ is the first or last in the history, called *firstEinH* and *lastEinH*:

```
isEmpty(ngb(property(crtGraph,node,now=="T"),port,viewLabel=="In"))
```

```
isEmpty(ngb(property(crtGraph,node,now=="T"),port,viewLabel=="Out"))
```

Regarding duties the first thing we need is to instantiate them. A duty is instantiated either in the beginning of a history if it is associated with an obligation such that $ge_1 = \perp$, or when the current event is an instantiation of ge and there is an obligation such that $ge_1 = ge$. The rewrite rules for the first case will be composed of five nodes and four edges on the left side and six nodes and seven edges on the right side. One of these rules is shown in Figure 5.16 where:

- L_W is defined by $N_L = \{n_1, n_2, n_3, n_4, n_5\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5\}$, $E_L = \{e_1, e_2, e_3, e_4\}$, $W = \{n_2\}$ and:

- $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, green)\};$
- $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, O), (viewLabel, v_4)\};$
- $\mathcal{L}_L(n_3) = \{(Interface, \{p_3\}), (ent, v_5), (type, P_r), (viewLabel, v_6), (viewColor, green)\};$
- $\mathcal{L}_L(n_4) = \{(Interface, \{p_4\}), (ent, v_7), (type, R), (viewLabel, v_8), (viewColor, green)\};$
- $\mathcal{L}_L(n_5) = \{(Interface, \{p_5\}), (ent, v_9), (type, A), (viewLabel, v_{10}), (viewColor, green)\};$
- $\forall p_i \in \{p_1, p_2, p_3\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, i)\};$
- $\forall p_i \in \{p_4, p_5\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\};$
- $\mathcal{L}_L(e_1) = \{(Connect, \{p_1, p_2\}), (type, PO), (aux, true), (viewColor, invisible)\};$
- $\mathcal{L}_L(e_2) = \{(Connect, \{p_2, p_3\}), (type, OP_r)\};$
- $\mathcal{L}_L(e_3) = \{(Connect, \{p_3, p_4\}), (type, P_rR)\};$
- $\mathcal{L}_L(e_4) = \{(Connect, \{p_3, p_5\}), (type, P_rA)\}.$

- R_M^N is defined by $N_R = \{n_1, n_2, n_3, n_4, n_5, n_6\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $E_R = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, $M = \{n_2\}$, $N = \{n_1\}$ and:

- $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i);$
- $\mathcal{L}_R(n_6) = \{(Interface, \{p_6\}), (ent, (v_1, v_9, v_7, \perp, na)), (type, D),$
 $(viewLabel, (v_2, v_{10}, v_8, \perp, na)), (state, pending), (viewColor, light\ blue)\};$
- $\forall p_i \in \{p_1, p_2, p_3\}, \mathcal{L}_R(p_i) = \{(Attach, n_i), (Arity, i + 1)\};$
- $\forall p_i \in \{p_4, p_5\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i);$
- $\mathcal{L}_R(p_6) = \{(Attach, n_6), (Arity, 3)\};$
- $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i);$
- $\mathcal{L}_R(e_5) = \{(Connect, \{p_1, p_6\}), (type, PD)\};$
- $\mathcal{L}_R(e_6) = \{(Connect, \{p_6, p_2\}), (type, DO), (aux, true), (viewColor, invisible)\};$
- $\mathcal{L}_R(e_7) = \{(Connect, \{p_6, p_3\}), (type, DP_r)\}.$

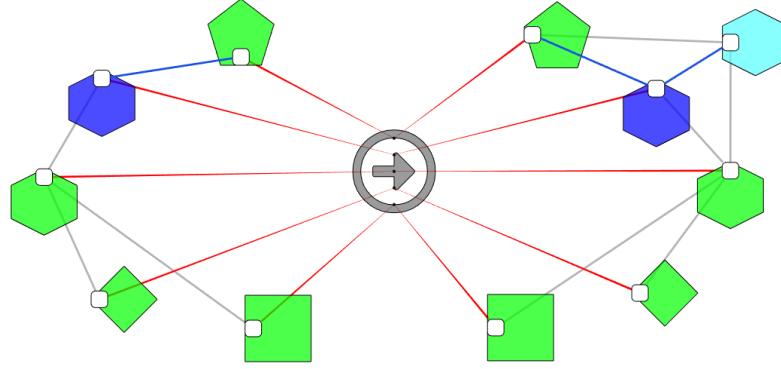


FIGURE 5.16: Rewrite rule instDbotggg

In this rule, as well as in the following rules in this example, every green node can also be blue, which means there will be sixteen versions of each of these rules. As before the invisible edges were selected for demonstrative purposes. The edge of type DO is an auxiliary edge that binds the created duty with the obligation associated with it, which will be useful later on. The strategy to apply these rules, called *instDbot*, is:

```

setBan(all(property(crtGraph,node,type=="0")\
    ngb(property(crtGraph,node,type=="G"),edge,type=="OG" && ge=="i")));
while(not(isEmpty(crtBan)))do(
    setPos(one(crtBan));
    setBan(all(crtBan\crtPos));
    repeat(#instDbotrr#);
    setBan(all(property(crtBan,node,type=="0")))
)

```

Where strategy *instDbotrr* is:

```
((((one(instDbotbbb))orelse(one(instDbotbbbg))
  )orelse((one(instDbotbbgb))orelse(one(instDbotbbgg)))
)orelse(((one(instDbotgbbb))orelse(one(instDbotgbgb))
  )orelse((one(instDbotgggb))orelse(one(instDbotgggg)))))
)orelse(((one(instDbotgbbb))orelse(one(instDbotgbgb))
  )orelse((one(instDbotgggb))orelse(one(instDbotgggg)))
)orelse(((one(instDbotgbbb))orelse(one(instDbotgbgb))
  )orelse((one(instDbotgggb))orelse(one(instDbotgggg)))))
```

The rewrite rules for the second case will be composed of seven nodes and six edges on the left side and eight nodes and ten edges on the right side. One of these rules is shown in Figure 5.17 where:

- L_W is defined by $N_L = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}\}$, $E_L = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, $W = \{n_2\}$ and $\mathcal{L}_L$ is the same as in the previous rule with $\mathcal{L}_L(p_2) = \{(Attach, n_2), (Arity, 3)\}$ and the following additions:
 - $\mathcal{L}_L(n_6) = \{(Interface, \{p_6, p_7, p_8\}), (ent, v_{11}), (type, E), (viewLabel, v_{12}), (now, \top)\}$;
 - $\mathcal{L}_L(n_7) = \{(Interface, \{p_9, p_{10}, p_{11}\}), (ent, v_{13}), (type, G), (viewLabel, v_{14})\}$;
 - $\mathcal{L}_L(p_6) = \{(Attach, n_6), (Arity, 1)\}$;
 - $\mathcal{L}_L(p_7) = \{(Attach, n_6), (Arity, 0), (viewLabel, In)\}$;
 - $\mathcal{L}_L(p_8) = \{(Attach, n_6), (Arity, 0), (viewLabel, Out)\}$;
 - $\mathcal{L}_L(p_9) = \{(Attach, n_7), (Arity, 2)\}$;
 - $\mathcal{L}_L(p_{10}) = \{(Attach, n_7), (Arity, 0), (viewLabel, In)\}$;
 - $\mathcal{L}_L(p_{11}) = \{(Attach, n_7), (Arity, 0), (viewLabel, Out)\}$;
 - $\mathcal{L}_L(e_5) = \{(Connect, \{p_2, p_9\}), (type, OG), (ge, i)\}$;
 - $\mathcal{L}_L(e_6) = \{(Connect, \{p_6, p_9\}), (type, EG), (aux, false)\}$.
- R_M^N is defined by $N_R = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}\}$, $E_R = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$, $M = \{n_2\}$, $N = \{n_1\}$ and:
 - $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;

- $\mathcal{L}_R(n_8) = \{(Interface, \{p_{12}\}), (ent, (v_1, v_9, v_7, v_{11}, na)), (type, D),$
 $(viewLabel, (v_2, v_{10}, v_8, v_{12}, na)), (state, pending), (viewColor, light\ blue)\};$
- $\forall p_i \in \{p_1, p_3\}, \mathcal{L}_R(p_i) = \{(Attach, n_i), (Arity, i + 1)\};$
- $\mathcal{L}_R(p_2) = \{(Attach, n_2), (Arity, 4)\};$
- $\mathcal{L}_R(p_6) = \{(Attach, n_6), (Arity, 1)\};$
- $\forall p_i \in P_L \setminus \{p_1, p_2, p_3, p_6\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i);$
- $\mathcal{L}_R(p_{12}) = \{(Attach, n_8), (Arity, 4)\};$
- $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i);$
- $\mathcal{L}_R(e_7) = \{(Connect, \{p_1, p_{12}\}), (type, PD)\};$
- $\mathcal{L}_R(e_8) = \{(Connect, \{p_{12}, p_2\}), (type, DO), (aux, true), (viewColor, invisible)\};$
- $\mathcal{L}_R(e_9) = \{(Connect, \{p_{12}, p_3\}), (type, DP_r)\};$
- $\mathcal{L}_R(e_{10}) = \{(Connect, \{p_{12}, p_6\}), (type, DE), (ev, i), (viewColor, green)\}.$

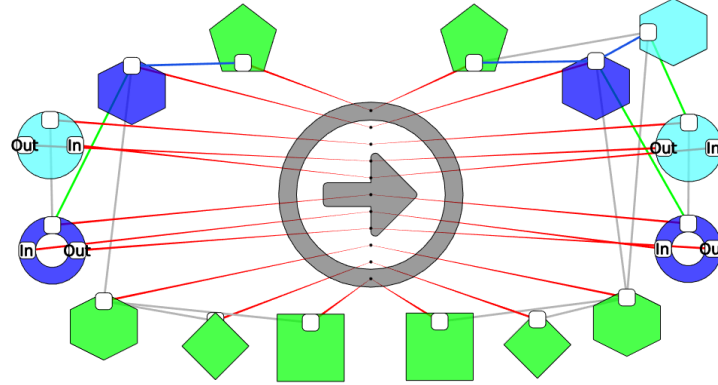


FIGURE 5.17: Rewrite rule instDggg

As you can see the edge of type EG has the value of attribute *aux* set to false, this is because we will need another version of these rules where this edge is auxiliary. One of these rules is shown in Figure 5.18, where the only difference to the previous one is:

- $\mathcal{L}_L(e_6) = \{(Connect, \{p_6, p_9\}), (type, EG), (aux, true), (viewColor, invisible)\}.$

The strategy to apply these rules, called *instD*, is:

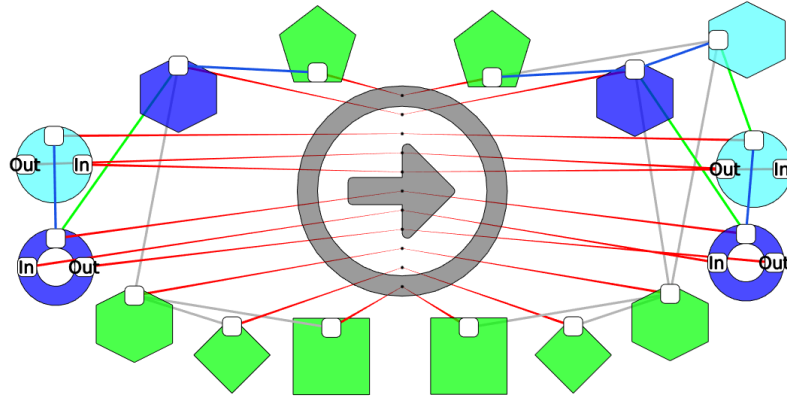


FIGURE 5.18: Rewrite rule instauxDgggg

```

setBan(all(property(crtGraph,node,type=="0")));
while(not(isEmpty(crtBan)))do(
  setPos(one(crtBan));
  setBan(all(crtBan\crtPos));
  repeat(#instDrr#);
  setBan(all(property(crtBan,node,type=="0")))
)

```

Where strategy *instDrr* is:

```

((((one(instDbbbb))orelse(one(instDbbbg))
  )orelse((one(instDbbgb))orelse(one(instDbbgg))))
  )orelse(((one(instDbgbb))orelse(one(instDbgbg))
    )orelse((one(instDbggb))orelse(one(instDbggg))))
)orelse((((one(instDgbbb))orelse(one(instDgbbg))
  )orelse((one(instDgbgb))orelse(one(instDgbgg))))
  )orelse(((one(instDggbg))orelse(one(instDggbg))
    )orelse((one(instDgggb))orelse(one(instDgggg))))
)orelse((((one(instauxDbbbb))orelse(one(instauxDbbbg))
  )orelse((one(instauxDbbgb))orelse(one(instauxDbbgg))))
  )orelse(((one(instauxDbgbb))orelse(one(instauxDbgbg))
    )orelse((one(instauxDbggb))orelse(one(instauxDbggg))))
)orelse((((one(instauxDgbbb))orelse(one(instauxDgbbg))
  )orelse((one(instauxDgbgb))orelse(one(instauxDgbgg))))
  )orelse(((one(instauxDggbg))orelse(one(instauxDggbg))
    )orelse((one(instauxDgggb))orelse(one(instauxDgggg))))

```


Because we do an evaluation of duties based on a history of events every duty will have attribute state. In both types of instantiation the value of the state attribute is set to pending. From this the duty can either go to fulfilled or violated. We will focus first on the transition from pending to violated. This happens when the current event is an instantiation of ge and there is an obligation such that $ge_2 = ge$ connected by an edge of type DO to a duty with the value of state set to pending. The first set of rewrite rules for this purpose is composed of nine nodes on both sides, eight edges on the left side and nine edges on the right side. One of these rules is shown in Figure 5.19 where:

- L_W is defined by $N_L = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}$, $E_L = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $W = \{n_6, n_8\}$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, green)\};$
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, O), (viewLabel, v_4)\};$
 - $\mathcal{L}_L(n_3) = \{(Interface, \{p_3\}), (ent, v_5), (type, P_r), (viewLabel, v_6), (viewColor, green)\};$
 - $\mathcal{L}_L(n_4) = \{(Interface, \{p_4\}), (ent, v_7), (type, R), (viewLabel, v_8), (viewColor, green)\};$
 - $\mathcal{L}_L(n_5) = \{(Interface, \{p_5\}), (ent, v_9), (type, A), (viewLabel, v_{10}), (viewColor, green)\};$
 - $\mathcal{L}_L(n_6) = \{(Interface, \{p_6, p_7, p_8\}), (ent, v_{11}), (type, E), (viewLabel, v_{12}), (now, \top)\};$
 - $\mathcal{L}_L(n_7) = \{(Interface, \{p_9, p_{10}, p_{11}\}), (ent, v_{13}), (type, G), (viewLabel, v_{14})\};$
 - $\mathcal{L}_L(n_8) = \{(Interface, \{p_{12}\}), (type, D), (state, pending)\};$
 - $\mathcal{L}_L(n_9) = \{(Interface, \{p_{13}, p_{14}, p_{15}\}), (ent, v_{15}), (type, E), (viewLabel, v_{16})\};$
 - $\forall p_i \in \{p_1, p_2, p_3\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, i)\};$
 - $\forall p_i \in \{p_4, p_5, p_6\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\};$
 - $\mathcal{L}_L(p_7) = \{(Attach, n_6), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_8) = \{(Attach, n_6), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(p_9) = \{(Attach, n_7), (Arity, 2)\};$
 - $\mathcal{L}_L(p_{10}) = \{(Attach, n_7), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_{11}) = \{(Attach, n_7), (Arity, 0), (viewLabel, Out)\};$

- $\mathcal{L}_L(p_{12}) = \{(Attach, n_8), (Arity, 4)\};$
 - $\mathcal{L}_L(p_{13}) = \{(Attach, n_9), (Arity, 1)\};$
 - $\mathcal{L}_L(p_{14}) = \{(Attach, n_9), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_{15}) = \{(Attach, n_9), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_3, p_4\}), (type, P_R)\};$
 - $\mathcal{L}_L(e_2) = \{(Connect, \{p_3, p_5\}), (type, P_A)\};$
 - $\mathcal{L}_L(e_3) = \{(Connect, \{p_2, p_9\}), (type, OG), (ge, f)\};$
 - $\mathcal{L}_L(e_4) = \{(Connect, \{p_6, p_9\}), (type, EG), (aux, false)\};$
 - $\mathcal{L}_L(e_5) = \{(Connect, \{p_1, p_{12}\}), (type, PD)\};$
 - $\mathcal{L}_L(e_6) = \{(Connect, \{p_{12}, p_2\}), (type, DO), (aux, true)\};$
 - $\mathcal{L}_L(e_7) = \{(Connect, \{p_{12}, p_3\}), (type, DP_r)\};$
 - $\mathcal{L}_L(e_8) = \{(Connect, \{p_{12}, p_{13}\}), (type, DE), (ev, i)\}.$
- R_M^N is defined by $N_R = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}$, $E_R = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$, $M = \{n_6\}$, $N = \{n_8\}$ and:
- $\forall n_i \in N_L \setminus n_8, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i);$
 - $\mathcal{L}_R(n_8) = \{(Interface, \{p_{12}\}), (ent, (v_1, v_9, v_7, v_{15}, v_{11})), (type, D), (viewLabel, (v_2, v_{10}, v_8, v_{16}, v_{12})), (state, violated), (viewColor, blue)\};$
 - $\forall p_i \in P_L \setminus \{p_6, p_{12}\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i);$
 - $\mathcal{L}_R(p_6) = \{(Attach, n_6), (Arity, 2)\};$
 - $\mathcal{L}_R(p_{12}) = \{(Attach, n_8), (Arity, 5)\};$
 - $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i);$
 - $\mathcal{L}_R(e_9) = \{(Connect, \{p_{12}, p_6\}), (type, DE), (ev, f), (viewColor, red)\}.$

The next set of rules for this purpose is almost equal to these with the only difference being that the edge connecting the current event to the event scheme is auxiliary:

- $\mathcal{L}_L(e_4) = \{(Connect, \{p_6, p_9\}), (type, EG), (aux, true)\}.$

One of these rules is shown in Figure 5.20.

Still with the same purpose in mind we need rules for when the duty in question is connected to an obligation such that $ge_1 = \perp$. Therefore the next set of rules is similar to the first one without node n_9 and correspondent ports and edge. One of these rules is shown in Figure 5.21 where:

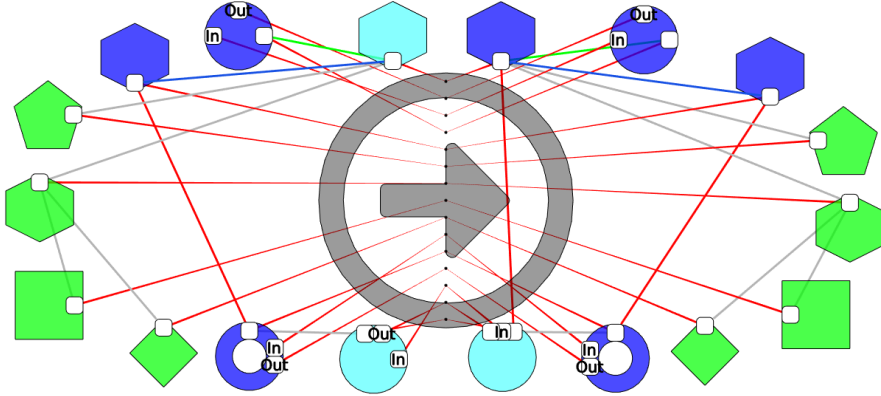


FIGURE 5.19: Rewrite rule updDpvgggg

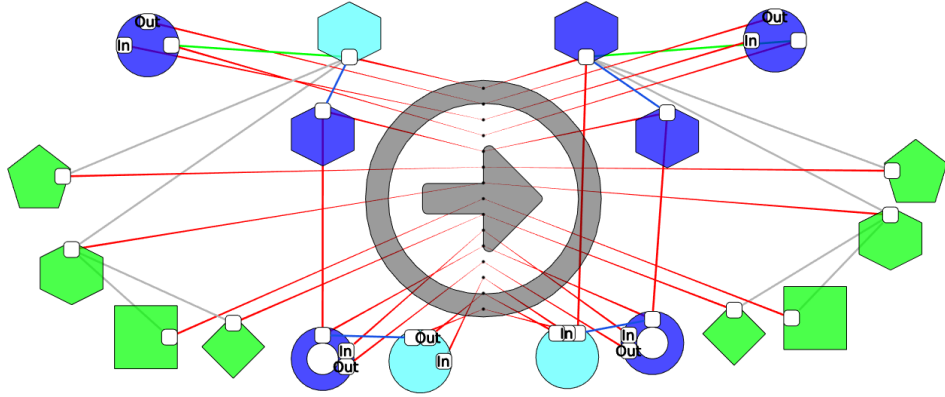


FIGURE 5.20: Rewrite rule updauxDpvgggg

- L_W is defined by $N_L = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}\}$, $E_L = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, $W = \{n_6, n_8\}$ and $\mathcal{L}_L$ is the same as in the first rule with $\mathcal{L}_L(p_{12}) = \{(Attach, n_8), (Arity, 3)\}$ and without the records for the dropped elements.
- R_M^N is defined by $N_R = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{10}, p_{11}, p_{12}\}$, $E_R = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $M = \{n_6\}$, $N = \{n_8\}$ and:
 - $\forall n_i \in N_L \setminus n_8, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
 - $\mathcal{L}_R(n_8) = \{(Interface, \{p_{12}\}), (ent, (v_1, v_9, v_7, \perp, v_{11})), (type, D), (viewLabel, (v_2, v_{10}, v_8, \perp, v_{12})), (state, violated), (viewColor, blue)\}$;
 - $\forall p_i \in P_L \setminus \{p_6, p_{12}\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i)$;
 - $\mathcal{L}_R(p_6) = \{(Attach, n_6), (Arity, 2)\}$;
 - $\mathcal{L}_R(p_{12}) = \{(Attach, n_8), (Arity, 4)\}$;

- $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i);$
- $\mathcal{L}_R(e_8) = \{(Connect, \{p_{12}, p_6\}), (type, DE), (ev, f), (viewColor, red)\}.$

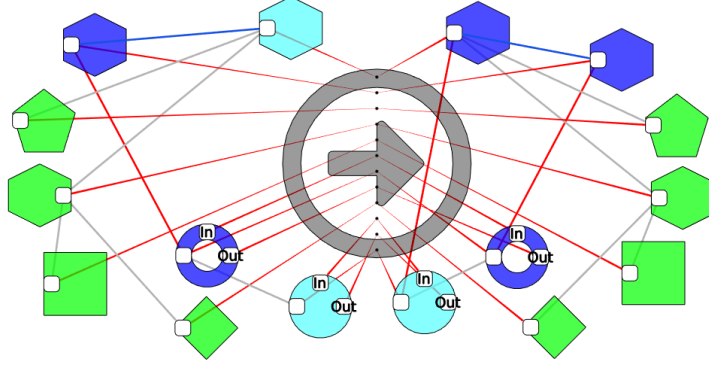


FIGURE 5.21: Rewrite rule updDpvbotggg

Finally, as with the first set of rules we need a version of these where the edge connecting the current event to the event scheme is auxiliary. One of these rules is shown in Figure 5.22.

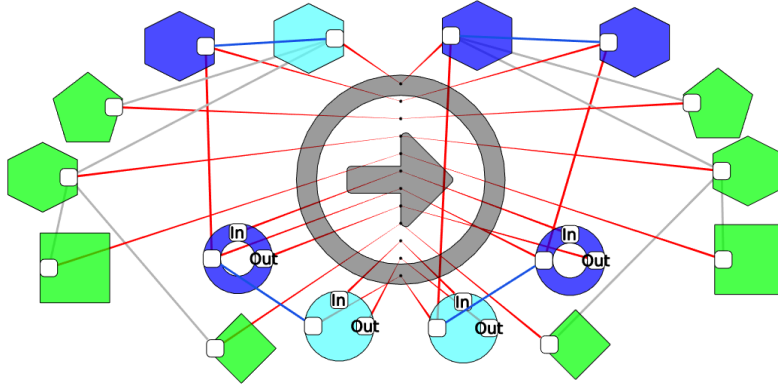


FIGURE 5.22: Rewrite rule updauxDpvbotggg

To apply these rules we use strategy *updDpv*:

```
setPos(all(property(crtGraph,node,type=="D" && state=="pending") [cup]
           property(crtGraph,node,now=="T")
));
repeat(#updDpvrr#);
repeat(#updDpvbotrr#)
```

Where strategy *updDpvrr* is:

```

((((one(updDpvbbbb))orelse(one(updDpvbbbg))
    )orelse((one(updDpvbbgb))orelse(one(updDpvbbgg)))
  )orelse(((one(updDpvbgbb))orelse(one(updDpvbgbg))
    )orelse((one(updDpvbggb))orelse(one(updDpvbggg)))))
)orelse((((one(updDpvgbbb))orelse(one(updDpvgbbg))
    )orelse((one(updDpvgbgb))orelse(one(updDpvgbgg)))
  )orelse(((one(updDpvggbb))orelse(one(updDpvggbg))
    )orelse((one(updDpvgggb))orelse(one(updDpvgggg)))))
)orelse((((one(updauxDpvbbbb))orelse(one(updauxDpvbbbg))
    )orelse((one(updauxDpvbbgb))orelse(one(updauxDpvbbgg)))
  )orelse(((one(updauxDpvbgbb))orelse(one(updauxDpvbgbg))
    )orelse((one(updauxDpvbggb))orelse(one(updauxDpvbggg)))))
)orelse((((one(updauxDpvgbbb))orelse(one(updauxDpvgbbg))
    )orelse((one(updauxDpvgbgb))orelse(one(updauxDpvgbgg)))
  )orelse(((one(updauxDpvggbb))orelse(one(updauxDpvggbg))
    )orelse((one(updauxDpvgggb))orelse(one(updauxDpvgggg)))))

```

And strategy updDpvbotrr is:

```

((((one(updDpvbotbbbb))orelse(one(updDpvbotbbbg))
    )orelse((one(updDpvbotbbgb))orelse(one(updDpvbotbbgg)))
  )orelse(((one(updDpvbotbgbb))orelse(one(updDpvbotbgbg))
    )orelse((one(updDpvbotbggb))orelse(one(updDpvbotbggg)))))
)orelse((((one(updDpvbotgbbb))orelse(one(updDpvbotgbbg))
    )orelse((one(updDpvbotgbgb))orelse(one(updDpvbotgbgg)))
  )orelse(((one(updDpvbotggbb))orelse(one(updDpvbotggbg))
    )orelse((one(updDpvbotgggb))orelse(one(updDpvbotgggg)))))
)orelse((((one(updauxDpvbotbbbb))orelse(one(updauxDpvbotbbbg))
    )orelse((one(updauxDpvbotbbgb))orelse(one(updauxDpvbotbbgg)))
  )orelse(((one(updauxDpvbotbgbb))orelse(one(updauxDpvbotbgbg))
    )orelse((one(updauxDpvbotbggb))orelse(one(updauxDpvbotbggg)))))
)orelse((((one(updauxDpvbotgbbb))orelse(one(updauxDpvbotgbbg))
    )orelse((one(updauxDpvbotgbgb))orelse(one(updauxDpvbotgbgg)))
  )orelse(((one(updauxDpvbotggbb))orelse(one(updauxDpvbotggbg))
    )orelse((one(updauxDpvbotgggb))orelse(one(updauxDpvbotgggg)))))

```

Now let's focus on the transition of duties from pending to fulfilled. This happens when the current event represents the action on the resource by the principal needed to fulfill the pending duty. The first set of rules for this purpose is composed of six nodes on both sides, seven edges on the left side and eight on the right. One of these rules is shown in Figure 5.23 where:

- L_W is defined by $N_L = \{n_1, n_2, n_3, n_4, n_5, n_6\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$, $E_L = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, $W = \{n_2, n_6\}$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, green)\}$;
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, D), (viewLabel, v_4), (state, pending)\}$;
 - $\mathcal{L}_L(n_3) = \{(Interface, \{p_3\}), (ent, v_5), (type, P_r), (viewLabel, v_6), (viewColor, green)\}$;
 - $\mathcal{L}_L(n_4) = \{(Interface, \{p_4\}), (ent, v_7), (type, R), (viewLabel, v_8), (viewColor, green)\}$;
 - $\mathcal{L}_L(n_5) = \{(Interface, \{p_5\}), (ent, v_9), (type, A), (viewLabel, v_{10}), (viewColor, green)\}$;
 - $\mathcal{L}_L(n_6) = \{(Interface, \{p_6, p_7, p_8\}), (ent, v_{11}), (type, E), (viewLabel, v_{12}), (now, \top)\}$;
 - $\forall p_i \in \{p_1, p_2, p_4, p_5\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 2)\}$;
 - $\forall p_i \in \{p_3, p_6\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 3)\}$;
 - $\mathcal{L}_L(p_7) = \{(Attach, n_6), (Arity, 0), (viewLabel, In)\}$;
 - $\mathcal{L}_L(p_8) = \{(Attach, n_6), (Arity, 0), (viewLabel, Out)\}$;
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_1, p_2\}), (type, PD)\}$;
 - $\mathcal{L}_L(e_2) = \{(Connect, \{p_2, p_3\}), (type, DP_r)\}$;
 - $\mathcal{L}_L(e_3) = \{(Connect, \{p_6, p_1\}), (type, EP)\}$;
 - $\mathcal{L}_L(e_4) = \{(Connect, \{p_3, p_4\}), (type, P_rR)\}$;
 - $\mathcal{L}_L(e_5) = \{(Connect, \{p_3, p_5\}), (type, P_rA)\}$;
 - $\mathcal{L}_L(e_6) = \{(Connect, \{p_6, p_4\}), (type, ER)\}$;
 - $\mathcal{L}_L(e_7) = \{(Connect, \{p_6, p_5\}), (type, EA)\}$.

-
- The diagram illustrates a network topology. At the center is a large grey circle containing a thick grey arrow pointing to the right, representing a router or a central node. Surrounding this central node are twelve smaller nodes, each represented by a different colored polygon: green pentagons, cyan hexagons, green squares, and cyan circles. These nodes are interconnected by a mesh of red lines, forming a complex network. A specific path is highlighted with a blue line, starting from the bottom-right cyan circle node, passing through the central router, and ending at the top-right cyan hexagon node. The bottom-left cyan circle node is labeled 'In' and the bottom-right cyan circle node is labeled 'Out'.

The diagram illustrates a network topology. At the center is a large grey circle containing a right-pointing arrow, representing a central router or hub. Surrounding this central node are twelve smaller nodes, each represented by a different colored polygon (green pentagon, cyan hexagon, blue hexagon, green pentagon, green hexagon, green square, green diamond, cyan circle, cyan circle, blue circle, cyan hexagon, green pentagon). These nodes are interconnected by a mesh of red lines, forming a complex network. Two specific nodes at the bottom are labeled 'In' and 'Out', indicating input and output points. The overall structure suggests a multi-protocol network environment.

FIGURE 5.24: Rewrite rule updDpfbotbotgggg


```

setPos(all(
  ngb((
    (property(crtGraph,node,type=="O")\
      ngb(property(crtGraph,node,type=="G"),edge,type=="OG" && ge=="f"))\
      ngb(property(crtGraph,node,type=="G"),edge,type=="OG" && ge=="i")
    ),edge,type=="DO")[cup]property(crtGraph,node,now=="T")
  ));
repeat(#updDpfbotbotrr#);
setPos(all(
  ngb((
    property(crtGraph,node,type=="O")\
      ngb(property(crtGraph,node,type=="G"),edge,type=="OG" && ge=="f")
    ),edge,type=="DO")[cup]property(crtGraph,node,now=="T")
  ));
repeat(#updDpfbotrr#);
setPos(all(property(crtGraph,node,type=="D" && state=="pending")[cup]
  property(crtGraph,node,now=="T")
));
repeat(#updDpfrr#)

```

Where strategy *updDpfbotbotrr* is:

```

(((one(updDpfbotbotbbbb))orElse(one(updDpfbotbotbbbg))
  )orElse((one(updDpfbotbotbbgb))orElse(one(updDpfbotbotbbgg)))
  )orElse(((one(updDpfbotbotbgbb))orElse(one(updDpfbotbotbgbg))
    )orElse((one(updDpfbotbotbggb))orElse(one(updDpfbotbotbggg))))
  )orElse(((one(updDpfbotbotgbbb))orElse(one(updDpfbotbotgbbg))
    )orElse((one(updDpfbotbotgbgb))orElse(one(updDpfbotbotgbgg)))
    )orElse(((one(updDpfbotbotggbb))orElse(one(updDpfbotbotggbg))
      )orElse((one(updDpfbotbotgggb))orElse(one(updDpfbotbotgggg))))))

```

And strategy *updDpfbotrr* (resp. *updDpfrr*) is obtained from this one by replacing every occurrence of "*updDpfbotbot*" with "*updDpfbot*" (resp. "*updDpf*").

Finally, we will need rules and strategies to update the color of fulfilled duties from light blue to blue. This happens when the current event is an instantiation of *ge* and there is an obligation

such that $ge_2 = ge$ connected by an edge of type *DO* to a duty with the value of state set to fulfilled and color set to light blue. The first set of rules for this purpose is composed of nine nodes in both sides, eight edges in the left side and nine edges in the right side. One of these rules is shown in Figure 5.26 where:

- L_W is defined by $N_L = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}$, $E_L = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $W = \{n_6, n_8\}$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, green)\};$
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_3), (type, O), (viewLabel, v_4)\};$
 - $\mathcal{L}_L(n_3) = \{(Interface, \{p_3\}), (ent, v_5), (type, P_r), (viewLabel, v_6), (viewColor, green)\};$
 - $\mathcal{L}_L(n_4) = \{(Interface, \{p_4\}), (ent, v_7), (type, R), (viewLabel, v_8), (viewColor, green)\};$
 - $\mathcal{L}_L(n_5) = \{(Interface, \{p_5\}), (ent, v_9), (type, A), (viewLabel, v_{10}), (viewColor, green)\};$
 - $\mathcal{L}_L(n_6) = \{(Interface, \{p_6, p_7, p_8\}), (ent, v_{11}), (type, E), (viewLabel, v_{12}), (now, \top)\};$
 - $\mathcal{L}_L(n_7) = \{(Interface, \{p_9, p_{10}, p_{11}\}), (ent, v_{13}), (type, G), (viewLabel, v_{14})\};$
 - $\mathcal{L}_L(n_8) = \{(Interface, \{p_{12}\}), (type, D), (state, fulfilled), (viewColor, light\ blue)\};$
 - $\mathcal{L}_L(n_9) = \{(Interface, \{p_{13}, p_{14}, p_{15}\}), (ent, v_{15}), (type, E), (viewLabel, v_{16})\};$
 - $\forall p_i \in \{p_1, p_2, p_3\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, i)\};$
 - $\forall p_i \in \{p_4, p_5, p_6\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\};$
 - $\mathcal{L}_L(p_7) = \{(Attach, n_6), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_8) = \{(Attach, n_6), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(p_9) = \{(Attach, n_7), (Arity, 2)\};$
 - $\mathcal{L}_L(p_{10}) = \{(Attach, n_7), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_{11}) = \{(Attach, n_7), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(p_{12}) = \{(Attach, n_8), (Arity, 4)\};$
 - $\mathcal{L}_L(p_{13}) = \{(Attach, n_9), (Arity, 1)\};$

- $\mathcal{L}_L(p_{14}) = \{(Attach, n_9), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(p_{15}) = \{(Attach, n_9), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_3, p_4\}), (type, P_rR)\};$
 - $\mathcal{L}_L(e_2) = \{(Connect, \{p_3, p_5\}), (type, P_rA)\};$
 - $\mathcal{L}_L(e_3) = \{(Connect, \{p_2, p_9\}), (type, OG), (ge, f)\};$
 - $\mathcal{L}_L(e_4) = \{(Connect, \{p_6, p_9\}), (type, EG), (aux, false)\};$
 - $\mathcal{L}_L(e_5) = \{(Connect, \{p_1, p_{12}\}), (type, PD)\};$
 - $\mathcal{L}_L(e_6) = \{(Connect, \{p_{12}, p_2\}), (type, DO), (aux, true)\};$
 - $\mathcal{L}_L(e_7) = \{(Connect, \{p_{12}, p_3\}), (type, DP_r)\};$
 - $\mathcal{L}_L(e_8) = \{(Connect, \{p_{12}, p_{13}\}), (type, DE), (ev, i)\}.$
- R_M^N is defined by $N_R = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}$, $E_R = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$, $M = \{n_6\}$, $N = \{n_8\}$ and:
 - $\forall n_i \in N_L \setminus n_8, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i);$
 - $\mathcal{L}_R(n_8) = \{(Interface, \{p_{12}\}), (ent, (v_1, v_9, v_7, v_{15}, v_{11})), (type, D), (viewLabel, (v_2, v_{10}, v_8, v_{16}, v_{12})), (state, fulfilled), (viewColor, blue)\};$
 - $\forall p_i \in P_L \setminus \{p_6, p_{12}\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i);$
 - $\mathcal{L}_R(p_6) = \{(Attach, n_6), (Arity, 2)\};$
 - $\mathcal{L}_R(p_{12}) = \{(Attach, n_8), (Arity, 5)\};$
 - $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i);$
 - $\mathcal{L}_R(e_9) = \{(Connect, \{p_{12}, p_6\}), (type, DE), (ev, f), (viewColor, red)\}.$

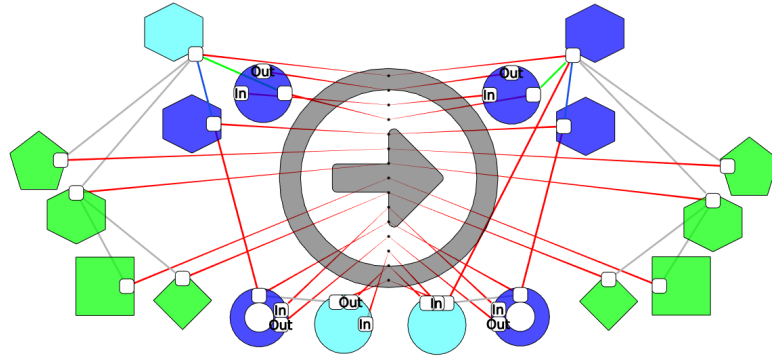


FIGURE 5.26: Rewrite rule upDffgggg

Because the edge of type EG can be auxiliary we will need another set of rules where the only difference to this is:

- $\mathcal{L}_L(e_4) = \{(\text{Connect}, \{p_6, p_9\}), (\text{type}, \text{EG}), (\text{aux}, \text{false})\}$.

One of these rules is shown in Figure 5.27.

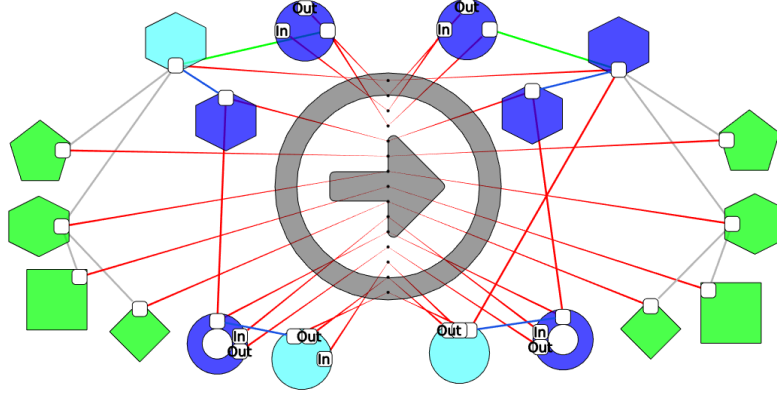


FIGURE 5.27: Rewrite rule updauxDffgggg

As the obligation in question can have $ge_1 = \perp$, we need another set of rules without node n_9 and respective ports and edge. One of these rules is shown in Figure 5.28 where:

- L_W is defined by $N_L = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}$, $P_L = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}\}$, $E_L = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, $W = \{n_6, n_8\}$ and $\mathcal{L}_L$ is the same as in the first rule with $\mathcal{L}_L(p_{12}) = \{(\text{Attach}, n_8), (\text{Arity}, 3)\}$ and without the records for the dropped elements.
- R_M^N is defined by $N_R = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}$, $P_R = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{10}, p_{11}, p_{12}\}$, $E_R = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $M = \{n_6\}$, $N = \{n_8\}$ and:

- $\forall n_i \in N_L \setminus n_8, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
- $\mathcal{L}_R(n_8) = \{(\text{Interface}, \{p_{12}\}), (\text{ent}, (v_1, v_9, v_7, \perp, v_{11})), (\text{type}, D), (\text{viewLabel}, (v_2, v_{10}, v_8, \perp, v_{12})), (\text{state}, \text{fulfilled}), (\text{viewColor}, \text{blue})\}$;
- $\forall p_i \in P_L \setminus \{p_6, p_{12}\}, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i)$;
- $\mathcal{L}_R(p_6) = \{(\text{Attach}, n_6), (\text{Arity}, 2)\}$;
- $\mathcal{L}_R(p_{12}) = \{(\text{Attach}, n_8), (\text{Arity}, 4)\}$;
- $\forall e_i \in E_L, \mathcal{L}_R(e_i) = \mathcal{L}_L(e_i)$;

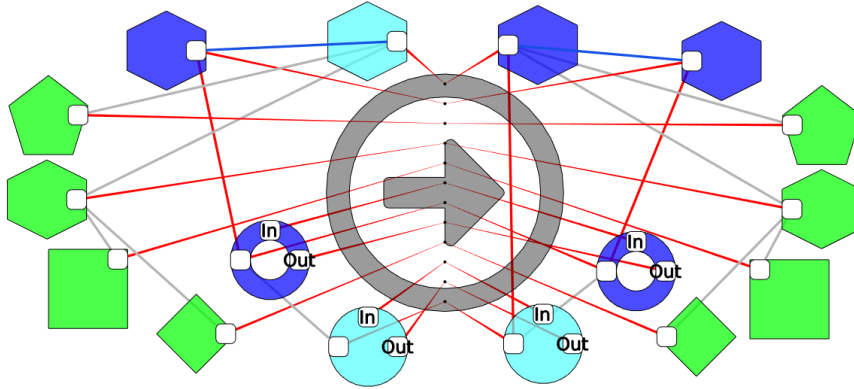


FIGURE 5.28: Rewrite rule updDffbotggg

$$- \mathcal{L}_R(e_8) = \{(Connect, \{p_{12}, p_6\}), (type, DE), (ev, f), (viewColor, red)\}.$$

Again, we will need another set of rules similar to these where the edge of type EG is auxiliary. One of these rules is shown in Figure 5.29.

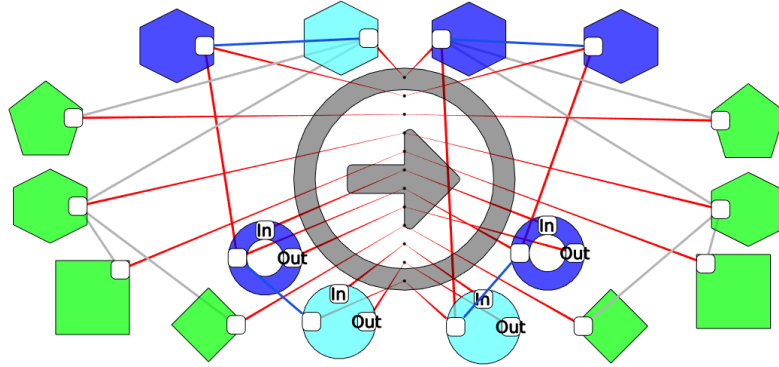


FIGURE 5.29: Rewrite rule updauxDffbotggg

To apply these rules we use strategy *updDff*:

```
setPos(all(
    property(crtGraph,node,type=="D" && state=="fulfilled")[cup]
    property(crtGraph,node,now=="T")
));
repeat(#updDffrr#);
repeat(#updDffbotrr#)
```

Where strategy *updDffrr* is:

And strategy `updDffbotrr` is:

```
(((((one(updDffbotbbbb))orelse(one(updDffbotbbbg))
    )orelse((one(updDffbotbbgb))orelse(one(updDffbotbbgg)))
    )orelse(((one(updDffbotbgbb))orelse(one(updDffbotbgbg))
    )orelse((one(updDffbotbggb))orelse(one(updDffbotbggg))))
    )orelse((((one(updDffbotgbbb))orelse(one(updDffbotgbbg))
    )orelse((one(updDffbotgbgb))orelse(one(updDffbotgbgg)))
    )orelse(((one(updDffbotggbg))orelse(one(updDffbotggbg))
    )orelse((one(updDffbotgggb))orelse(one(updDffbotgggg))))))
    )orelse((((one(updauxDffbotbbbb))orelse(one(updauxDffbotbbbg))
    )orelse((one(updauxDffbotbbgb))orelse(one(updauxDffbotbbgg)))
    )orelse(((one(updauxDffbotbgbb))orelse(one(updauxDffbotbgbg))
    )orelse((one(updauxDffbotbggb))orelse(one(updauxDffbotbggg))))
    )orelse((((one(updauxDffbotgbbb))orelse(one(updauxDffbotgbbg))
    )orelse((one(updauxDffbotgbgb))orelse(one(updauxDffbotgbgg)))
    )orelse(((one(updauxDffbotggbg))orelse(one(updauxDffbotggbg))
    )orelse((one(updauxDffbotgggb))orelse(one(updauxDffbotgggg))))))
```

With everything we discussed up to here done, we use strategy *calcD* to actually calculate duties:

```
#auxEG#;#auxPO#;
if(#firstEinH#)then(#instDbot#);
while(not(#lastEinH#))do(
  #updD#;#nextE#
);
#updD#;#closeH#;
setPos(all(crtGraph));#rmauxEG#;#rmauxPO#
```

Where strategies *auxEG* and *auxPO* (resp. *rmauxEG* and *rmauxPO*) create (resp. remove) the needed auxiliary edges of type EG and PO respectively, and strategy *updD* is:

```
#updDff#;#updDpv#;#updDpf#;#instD#
```

5.4 Views and Queries

In this section we will present some of the rewrite rules and strategies needed to show different possible views of the graph and queries of the policy. In terms of rewrite rules all we need is to have rules that hide and show all the different types of nodes and edges. Having these rules we use strategies to control which of them are applied, sometimes coupled with strategies discussed in the previous sections, to get the desired views. We will need to make rewrite rules and strategies to hide/show:

- nodes of type: P, C, P_r, A, R, O, D, E and G ;
- edges of type: $PC, PP_r, PO, PD, CC, CP_r, CO, P_rA, P_rR, OP_r, OG, DP_r, DE, DO, GG, EP, EA, ER, EG$ and EE .

Example 5.5. The rewrite rules to hide/show nodes will be composed of one node in each side, one of which invisible according to the purpose of the rule. In Figure 5.30 rewrite rule *hidePy* is shown, where:

- L_W is defined by $N_L = \{n_1\}, P_L = \{p_1\}, E_L = \emptyset, W = \emptyset$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, yellow)\}$;

- $\mathcal{L}_L(p_1) = \{(Attach, \{n_1\}), (Arity, 0)\}$.
- R_M^N is defined by $N_R = \{n_1\}$, $P_R = \{p_1\}$, $E_R = \emptyset$, $M = R$, $N = \emptyset$ and:
 - $\mathcal{L}_R(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, "invisible\ yellow"), (viewBorderColor, invisible), (viewLabelColor, invisible), (viewLabelBorderColor, invisible)\}$;
 - $\mathcal{L}_R(p_1) = \{(Attach, \{n_1\}), (Arity, 0), (viewColor, invisible), (viewBorderColor, invisible)\}$.

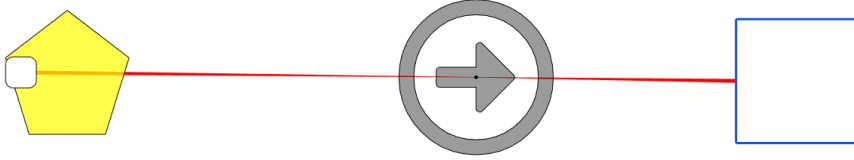


FIGURE 5.30: Rewrite rule hidePy

As before we have selected the invisible node for demonstrative purposes. The invisibility is accomplished by setting the alpha component of the color to zero, that is why the viewColor value of the right node is set to "invisible yellow". This way the "color" of the node is preserved when making it invisible and we can use this "color" when turning it visible again. In Figure 5.31 rewrite rule showPy is shown, where:

- L_W is defined by $N_L = \{n_1\}$, $P_L = \{p_1\}$, $E_L = \emptyset$, $W = \emptyset$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, "invisible\ yellow")\}$;
 - $\mathcal{L}_L(p_1) = \{(Attach, \{n_1\}), (Arity, 0)\}$.
- R_M^N is defined by $N_R = \{n_1\}$, $P_R = \{p_1\}$, $E_R = \emptyset$, $M = R$, $N = \emptyset$ and:
 - $\mathcal{L}_R(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, yellow)\}$;
 - $\mathcal{L}_R(p_1) = \{(Attach, \{n_1\}), (Arity, 0)\}$.

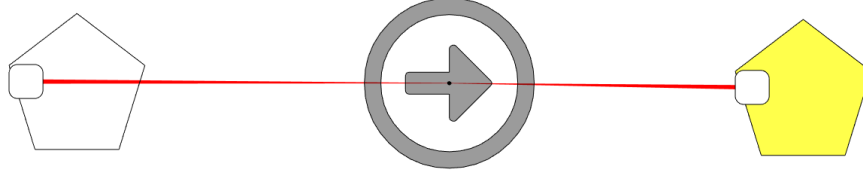


FIGURE 5.31: Rewrite rule showPy

In this rewrite rule we can see the border of the invisible node because only the value of *viewColor* is used to match possible nodes. As before we will need three versions of these rules, one for each possible color of the nodes. The strategy used to hide all nodes of type *P*, called *hideP*, is:

```
repeat(one(hidePb));
repeat(one(hidePg));
repeat(one(hidePy))
```

The one to show them, called *showP*, is equal to this with the only difference being the names of the rules:

```
repeat(one(showPb));
repeat(one(showPg));
repeat(one(showPy))
```

As the color of nodes holds information we have strategy *hidePp* (resp. *hidePo*) that hides nodes belonging only to the permission (resp. obligation) subgraph by only hiding yellow (resp. blue) nodes of type *P*. The same goes for strategies to show them.

The rewrite rules to hide/show edges will be composed of two nodes and one edge in both sides, one of the edges invisible depending on the purpose of the rule. In Figure 5.32 rewrite rule *hidePCyy* is shown, where:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2, p_3, p_4\}$, $E_L = \{e_1\}$, $W = \emptyset$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_2), (viewColor, yellow)\}$;

- $\mathcal{L}_L(n_2) = \{(Interface, \{p_2, p_3, p_4\}), (ent, v_3), (type, C), (viewLabel, v_4), (viewColor, yellow)\};$
 - $\forall p_i \in \{p_1, p_2\}, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\};$
 - $\mathcal{L}_L(p_3) = \{(Attach, n_2), (Arity, 0), (viewLabel, Out)\};$
 - $\mathcal{L}_L(p_4) = \{(Attach, n_2), (Arity, 0), (viewLabel, In)\};$
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_1, p_2\}), (type, PC), (viewColor, grey)\}.$
- R_M^N is defined by $N_R = \{n_1, n_2\}, P_R = \{p_1, p_2, p_3, p_4\}, E_R = \{e_1\}, M = R, N = \emptyset$ and:
 - $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i);$
 - $\forall p_i \in P_L, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i);$
 - $\mathcal{L}_R(e_1) = \{(Connect, \{p_1, p_2\}), (type, PC), (viewColor, invisible)\}.$

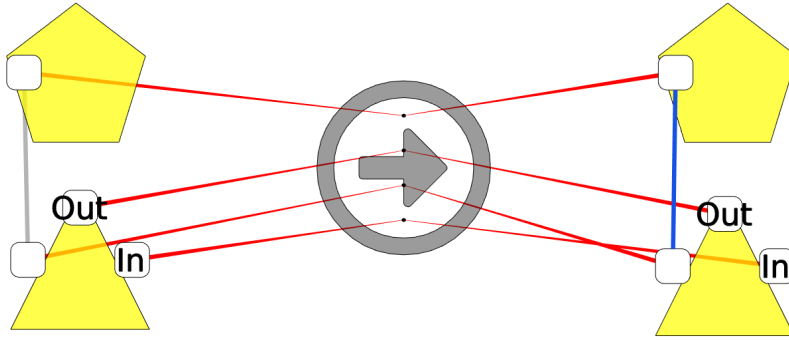


FIGURE 5.32: Rewrite rule hidePCyy

By switching the values of the *viewColor* attribute of the edges we get rewrite rule *showPCyy*, shown in Figure 5.33.

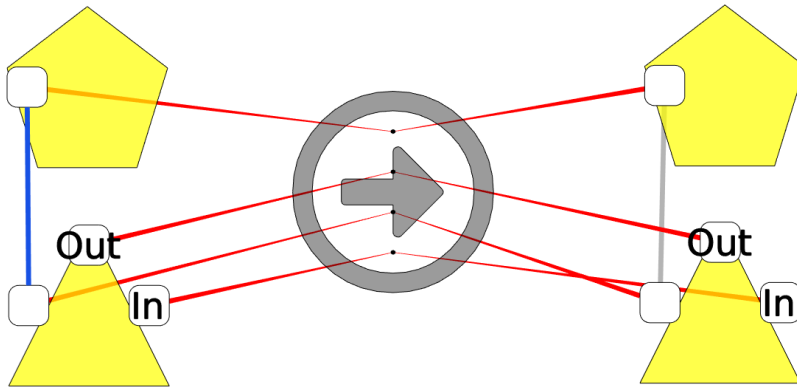


FIGURE 5.33: Rewrite rule showPCyy

*As before we have to make a version of these two rules for every possible combination of colors the nodes can be. As they can both be yellow, green or blue we would supposedly have to create nine versions of both rules but, as a blue node will never be connected to a yellow one, there will only be seven versions of each rule. Because the nodes are visible in these rules when we want to hide (resp. show) both edges and nodes we will hide (resp. show) all the edges (resp. nodes) first and only then all the nodes (resp. edges). The strategy to hide every edge of type PC, called *hidePC*, is:*

```
repeat(one(hidePCyy));repeat(one(hidePCyg));repeat(one(hidePCgy));
repeat(one(hidePCgg));repeat(one(hidePCgb));repeat(one(hidePCbg));
repeat(one(hidePCbb))
```

*The one to show them, called *showPC*, is:*

```
repeat(one(showPCyy));repeat(one(showPCyg));repeat(one(showPCgy));
repeat(one(showPCgg));repeat(one(showPCgb));repeat(one(showPCbg));
repeat(one(showPCbb))
```

*As with nodes we make strategies to hide/show only edges that belong to the permission (resp. obligation) subgraph by only using rules with yellow (resp. blue) nodes. Such a strategy, called *hidePCp*, is:*

```
repeat(one(hidePCyy));repeat(one(hidePCyg));repeat(one(hidePCgy))
```

*After making the rewrite rules and strategies to hide/show all types of nodes and edges we can compose them to get views of the graph. If we want to see the permission subgraph we can use strategy *hideObl*:*

```
#hidePCo#;#hidePD#;#hideCCo#;#hideCO#;#hidePrAo#;#hidePrRo#;#hideOPr#;
#hideOG#;#hideDPr#;#hideDE#;#hideDO#;#hideGG#;#hideEP#;#hideEA#;
#hideER#;#hideEG#;#hideEE#;
#hidePo#;#hideCo#;#hidePro#;#hideO#;#hideD#;#hideAo#;#hideRo#;#hideG#;
#hideE#
```

*The strategy *showObl* used to show the obligation subgraph after hiding it, is obtained from this one by substituting every "hide" with "show" and reversing the order of application of the strategies. There will be strategies *hideAuth* and *showAuth* that hide/show the permission subgraph.*

Consider the CBACO policy presented in Example 4.2 with its graphical representation shown again in Figure 5.34. The result of applying strategy *hideObl* (resp. *hideAuth*) to this one is shown in Figure 5.35 (resp. Figure 5.36).

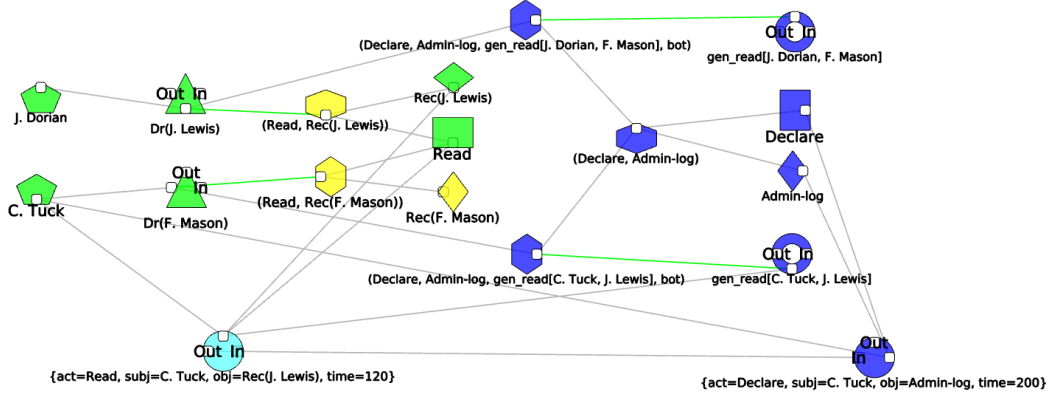


FIGURE 5.34: CBACO policy representation

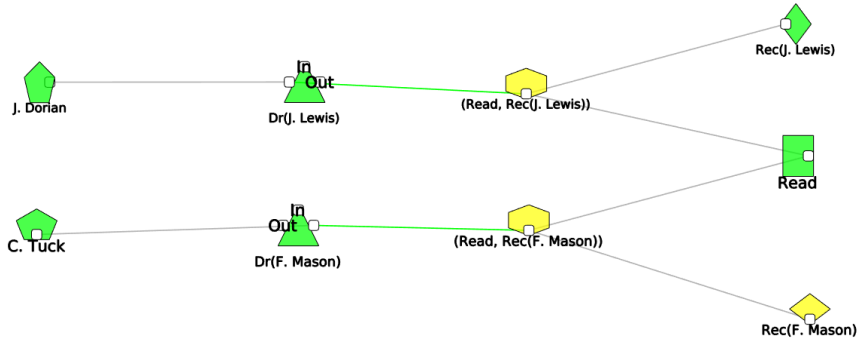


FIGURE 5.35: Result of applying strategy *hideObl*

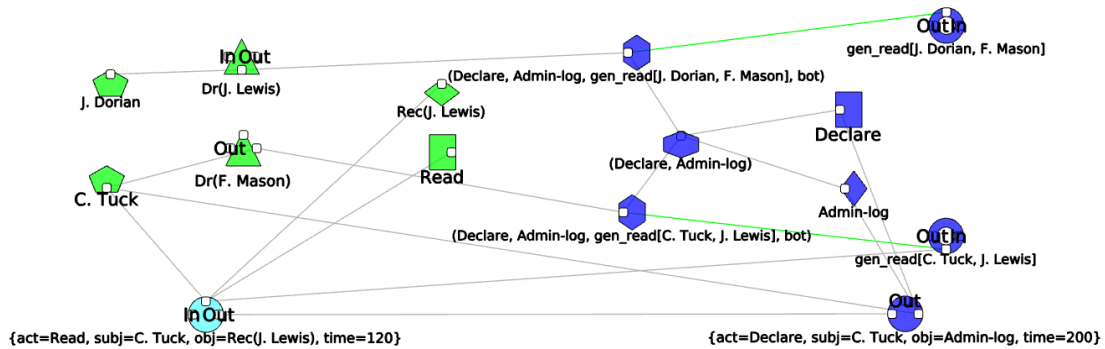


FIGURE 5.36: Result of applying strategy *hideAuth*

Now consider the CBAC policy described in Example 5.3 with its graphical representation shown again in Figure 5.37. Using strategy *auxPPru*, also described in Example 5.3, coupled with hiding strategies we build strategy *Undet* that calculates and shows the *UNDET* relation.

```
#auxPPru#;
setPos(all(crtGraph));
#hidePC#;#hidePD#;#hideCC#;#hideCPr#;#hideCO#;#hidePrA#;#hidePrR#;
#hideOPr#;#hideOG#;#hideDPr#;#hideDE#;#hideDO#;#hideGG#;#hideEP#;#hideEA#;
#hideER#;#hideEG#;#hideEE#;
setBan(all(property(crtGraph,edge,type=="PPr"&&auth=="U")));
#hideP#;#hidePr#;
setBan(all([emptySet]));
#hideC#;#hideO#;#hideD#;#hideA#;#hideR#;#hideG#;#hideE#
```

In Figure 5.38 we show the result of applying this strategy to the graph in Figure 5.37.

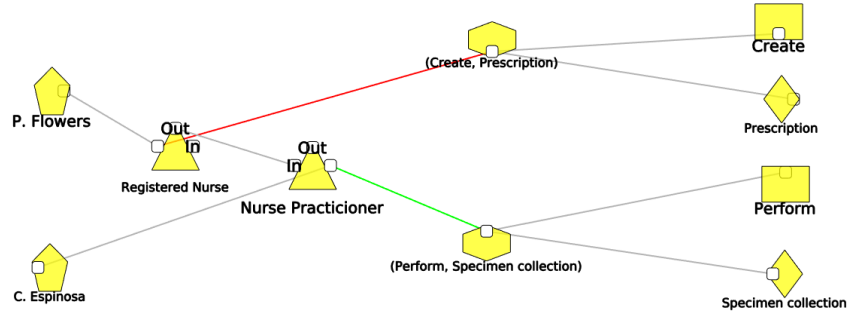


FIGURE 5.37: CBAC policy representation

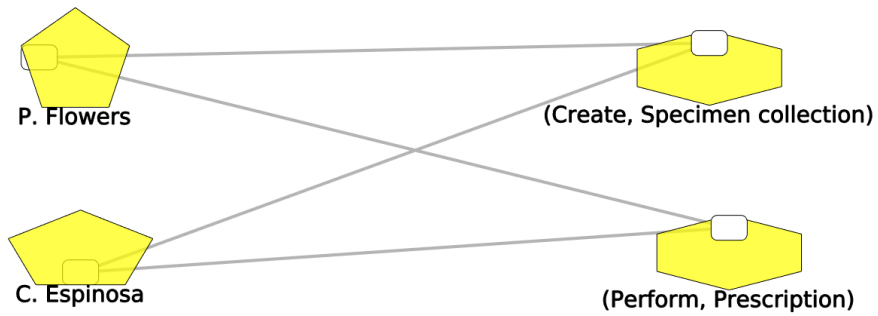


FIGURE 5.38: Result of applying strategy Undet

With this, we conclude the presentation of our implementation of the CBACO graph-based representation framework in PORGY. In the following chapters we discuss related work and make some concluding remarks.

Chapter 6

Related work

Most work done on CBAC policies uses only textual languages and focuses on the expressiveness of the model, the analysis of policies and techniques that can be used to enforce policies [1, 9, 16–18]. Graph based models for the analysis of CBAC policies, were introduced in [3, 12]. In [12] a graph-based framework for the representation and analysis of CBAC policies is defined, which was extended in [3] to CBAC policies with prohibitions. These works, which were described in detail in the background, form the basis for this thesis but do not address CBACO policies. Beyond the representation of policies these papers focus on the general dynamics associated with CBAC policies and specifically on the case of emergency management. Based on the model presented in [12], two implementations can be found in [19, 20]. Policy Manager [19] is an application for the creation, visualization and analysis of CBAC policies. It lets you add principals, categories, actions and resources, as well as define category-principal and permission-category assignments. The representation is graph based and similar to ours with only nodes of type P , C and P_r . There is also a window where one can get the answer to simple queries. G-ACM [20] is a graphic web based tool that uses a rule engine to implement the dynamics of the model described in [12]. The tool is composed of two parts, a console and an engine. In the console there is a static representation of the current status of the policy and, among other things, we can select from a variety of facts that, following an update request, will be fed to the engine in order to calculate the resulting policy graph given such facts. G-ACM representation of policies is based on the CBAC metamodel and, therefore, policies represented there are easily mapped into our model. The facts are the same as events in our model and, as so, we can also implement the dynamic part of the tool using our model. In addition to the main feature G-ACM offers a number of useful viewing features, most of

which could be integrated into our implementation with the use of strategies and rewrite rules. These include the possibility of hiding/showing edges that correspond to specific relations in the model, highlighting of nodes and edges relevant to a specific node and grouping of similar nodes (nodes that have the same relevant connections). One other viewing feature is the possibility to highlight in the graph the changes between the current policy and a previous instance of it, while this cannot be done in PORGY one can simply view the two instances simultaneously and compare them visually.

The use of graph based languages to model and analyse policies has mostly been developed in the context of RBAC [21–25]. In the following, we will describe some of these tools and models in more detail. In [23, 24], inspired by Nyanchama and Osborn in [22], Koch et al. use directed graphs to analyse RBAC policies using a graph model similar to Baldwin’s privilege graphs [21]. Privilege graphs are three layered acyclic directed graphs (one layer for users, one for roles and one for privileges) that are similar to the CBAC part of our model without the nodes of type A and R and respective edges. One key aspect of [23, 24] is the use of graph transformations to model administrative operations on RBAC policies. This focus on administrative operations comes from the fact that RBAC policies are static, whereas in CBAC the same operations are dependant on the system state and are carried out without the need for an administrator to intervene (an example of this behaviour can be seen in Section 5.1). Using our model we could represent the RBAC policies in [23, 24] since roles are a particular case of category; however, the graphs in [23, 24] also deal with sessions. In this thesis we have not dealt with sessions, but since the notion of session in RBAC is similar to the notion of session in CBAC it would be easy to adapt the formalism used in [23, 24] to our model. In [26, 27] a framework for the description of access control models using graph transformations is described. With each type of access control model is associated a typegraph for its graph based representation. Graph transformations are used to describe both the ways that the policy can evolve as well as positive and negative constraints. One of the focus of the work is the integration of policies that use different access control models. As the CBAC metamodel has been shown to subsume many of the most well known access control models it makes it a good candidate for this purpose.

In [28] three modeling notations (viz. UML, Alloy and Graph Transformations) are compared for the task of specification and verification of RBAC models. In the paper the focus is on the evolution and constraints of the policies and, with it, the possible violation

of constraints that may arise. Although we have not dealt with constraints in our implementation it would be easy to integrate them. In [25] conceptual graphs and conceptual graph rules are used to represent and reason on RBAC policies. Conceptual graphs are just labelled hypergraphs, meaning edges (called hyperedges) can represent any n-ary relationship between nodes. Rules are used to derive authorisation information from the policy graph and to formulate constraints. One interesting aspect of the paper is the use of backwards chaining to find inconsistencies in the policy; starting from a rule defining a constraint they work their way backwards (using the rules of information derivation in the opposite direction) to find if the policy could violate said constraint. While such mechanism is not possible in PORGY we could still check for the same constraint violations.

LaSCO [29] is a policy language based on graphs. The graphs are labelled; nodes represent objects (users, classes, resources, etc) and directed edges represent actions (events in the LaSCO terminology). A policy graph in LaSCO describes both the system state to which it applies and the constraints on that state for it to be valid; this is done with the use of labels (predicates in the LaSCO terminology). These labels can be seen as type of categorisation and so we can map these policy graphs into policies in our model.

Miró [30] is a set of languages and tools for visual specification of file system security inspired by Harel's work on higraphs [31]. Higraphs are a visual formalism of topological nature that combine Euler/Venn diagrams with hypergraphs. Instead of nodes we have blobs that can contain/intersect other blobs and can be connected by hyperedges making it a powerful formalism to represent a wide variety of complex systems. In the case of Miró, blobs are used to represent sets of users and files, and binary labeled directed edges describe the granting/denial of access rights or, in some cases (constraint language), if a blob is or is not inside another blob. There are two languages, an instance language that describes the access rights of users to files, and the constraint language that restricts the set of allowed instance pictures. The blobs representing sets of users are a kind of categorisation and as so this part of the representation is easily translated into our model. The blobs representing sets of files is something we cannot directly translate to our model as it is right now, but could easily be integrated as a partial ordering of resources such as with categories. As stated edges can be used to represent denial of access rights as in our model but, in contrast to our model, these denials propagate from the more general blobs to the less general as with granting. Furthermore, the lack of an edge granting some

user access to a file is interpreted as denial of access, and there can be multiple edges that apply to the same triple (user, action, file), in which case a predominant edge is searched. If there is no such edge it classifies as an ambiguity (in our model such ambiguities are not allowed by definition). Because of these differences, mapping a higraph in the Miró languages to a CBACO policy graph is not trivial, nonetheless, policies described using the Miró languages could be represented using our model.

In [32] graphs are used to define a framework for the representation of various access control models. Based on this framework a visual language for the definition of constraints is also defined. Another focus of the paper is on triggers, events that change the state of the policy or system. With some adaptation in some cases, every model, constraint and trigger discussed could be translated into our model.

Chapter 7

Conclusion

This thesis presents a graph-based metamodel that deals with obligations in access control. We implement the model in the PORGY framework, taking advantage of the port-graph representation and the port-graph rewriting based language. The strategy driven language of PORGY is powerful enough to deal with all the dynamic aspects of the proposed model. There are however some aspects that we would like to improve in the future. Presently there is a huge repetition of rules that result from the need to define the rules for all the nodes of the same type that may differ in their attributes (for example, its colour). Although this leads to a rather cumbersome translation of the policy graphs into PORGY, this aspect could be significantly improved by automating the generation of duplicated graphs/rules. In fact it should be possible to automate the creation of all the general rewrite rules, with one interesting addition being letting the user define additional attributes for each type of node. Something that would drastically reduce the number of rules would be the possibility of assigning value variables to the color attribute of elements in rules, which we do with other attributes but is not possible with color values in PORGY. Another improvement that would ease the task of policy design includes defining user-friendly Domain Specific Languages (DSLs), suitable for policy administrators that can be translated into our modelling languages or other programming languages for integration with existing policy analysis tools. Additionally, the use of higher-order notions of port-graphs could facilitate the abstraction of patterns in the policy graphs, which would lead to more efficient ways of writing rules that depend on information on nodes that are not directly connected. Another useful addition would be a layout algorithm for these representations as the ones present in PORGY are not well suited to make the policy graphs readable.

Rewriting systems have the advantages of being multi-paradigm and having a well-developed theory, and furthermore, they allow the use of languages and tools such as MAUDE, TOM and CiME for expeditious prototyping and analysis. For example, the rewrite-based operational semantics of the CBACO metamodel [2], allows for the analysis of compatibility between authorisations and obligations. We would like to further develop this analysis and use our graphical model and the strategy driven language of PORGY to analyse additional properties related to obligations, such as accountability.

CBAC is a fairly simple model, the addition of obligations makes it a little more complex, and the axiomatization of the state of duties was not straightforward. In the design of both our model and our implementation of it, the choices that were made always sought to augment the expressiveness and robustness of the model.

PORGY is in constant development. This means that between the time our implementation was realized and the writing of this paper features were added to it that our implementation does not take advantage of. It also means that sometimes things do not work as expected, and in our case there were a few times where this was caused by bugs in PORGY, fortunately there is a debug mode that lets you understand what is going on in terms of rewrite rule application, where you can actually follow the code finding morphisms between graphs, and when that did not help the developer Bruno Pinaud was always reachable to help find the problem. The fact that PORGY is in constant development also opens the door for possible additions being made that would directly benefit our needs.

Appendix A

Importing policy graphs into PORGY

To represent a policy graph in PORGY [4] one could simply set an empty graph and start adding nodes and edges one by one. Alternatively, it is possible to do so using Tulip [6] to import graph elements from CSV files, a couple of rewrite rules and strategies, and a small Python [33] script. This method is what we are going to show in this appendix.

A.1 CSV import in Tulip

To import graph elements into Tulip using CSV files we need at least two files, one containing the information of nodes and one containing the information of edges. In Figure A.1 we show an example of a CSV file with information on nodes. Note that we put only the necessary attributes: *ent* and *type* for every node and *now* for nodes of type *E*. Additionally nodes have attribute *id*, this is used in the CSV file with edges, shown in Figure A.2, to show which nodes edges connect to using the *Source* and *Target* fields.

Having these files all we need to do is use the feature CSV in Tulip to import the nodes and then the edges making sure to select property *id* in the Source and Destination properties when importing the edges as seen on Figure A.3.

Doing this with our example files we get the graph shown in Figure A.4. After this we export the graph and save it with the extension ".porgy.tlp" and we end up with a graph we can use in PORGY.

```

id,ent,type,now
1,J. Dorian,P,
2,C. Tuck,P,
3,Dr(J. Lewis),C,
4,Dr(F. Mason),C,
5,"(Read, Rec(J. Lewis))",MN,
6,"(Read, Rec(F. Mason))",MN,
7,"(Declare, Admin-log, gen_read[C. Tuck, J. Lewis], bot)",0,
8,"(Declare, Admin-log, gen_read[J. Dorian, F. Mason], bot)",0,
9,Read,A,
10,Declare,A,
11,Rec(J. Lewis),R,
12,Rec(F. Mason),R,
13,Admin-log,R,
14,"gen_read[C. Tuck, J. Lewis]",G,
15,"gen_read[J. Dorian, F. Mason]",G,
16,"(Declare, Admin-log)",MN,
17,"{act=Read, subj=C. Tuck, obj=Rec(J. Lewis), time=120}",E,T
18,"{act=Declare, subj=C. Tuck, obj=Admin-log, time=200}",E,

```

FIGURE A.1: Example of CSV file with nodes

```

Source,Target,type,auth,ge,target
1,3,PC,,,
2,4,PC,,,
3,5,CMN,A,,
4,6,CMN,A,,
3,8,CO,,,
4,7,CO,,,
5,9,MNA,,,
6,9,MNA,,,
5,11,MNR,,,
6,12,MNR,,,
7,16,OMN,,,
8,16,OMN,,,
7,14,OG,,i,
8,15,OG,,i,
16,10,MNA,,,
16,13,MNR,,,
17,14,EG,,,
17,18,EE,,, "{act=Declare, subj=C. Tuck, obj=Admin-log, time=200}"
17,9,EA,,,
17,2,EP,,,
17,11,ER,,,
18,10,EA,,,
18,13,ER,,,
18,2,EP,,,

```

FIGURE A.2: Example of CSV file with edges

Import CSV data into current graph: graph_0

Import method

Choose and configure the import method.

Import data on

New entities (nodes)

New relations (edges)

Existing entities (nodes)

Existing relations (edges)

Import data on new relations (edges)

Create a new relation (edge) for each row between a source entity(node) and a destination entity(node) and import the row data of the columns selected in the **Data selection** page on the created relation.

Source columns

Destination columns

Source

Target

The above selected columns specify the identifiers of the source and destination entities.
For each row, values in the 'Source/Destination columns' are compared, respectively, to the values in the 'Source/Destination properties' (see below) for existing node entities and mapped to the first matching node.

Source properties

Destination properties

id

id

Add new property

When no node entity is found, a new node is created, if the "Create missing entities" option below is checked.

☐ Create missing entities

< Back

Import into current graph

Cancel

FIGURE A.3: Tulip CSV import window

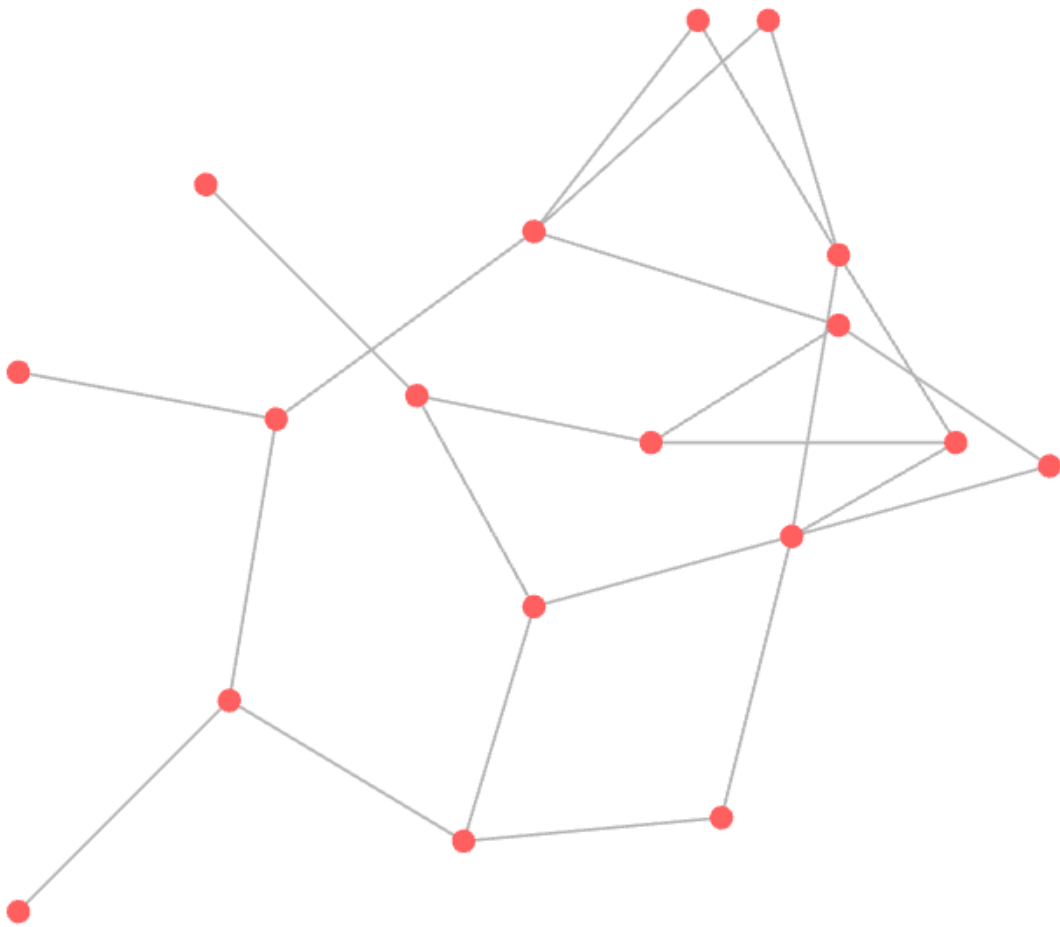


FIGURE A.4: Policy graph in Tulip

A.2 Getting the graph ready in PORGY

Importing the file we exported from Tulip using the Porgy Tlp import plugin in PORGY (making sure we select "false" in "Additional ports" as seen in Figure A.5) we get the graph shown in Figure A.6.

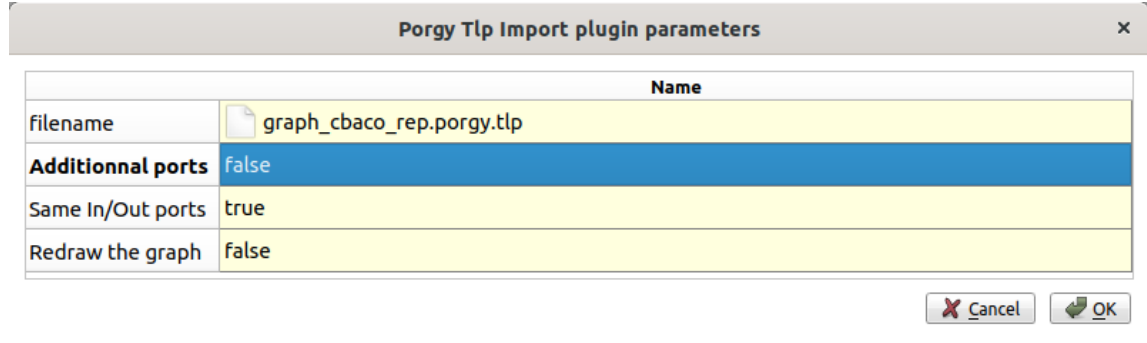


FIGURE A.5: Porgy Tlp import

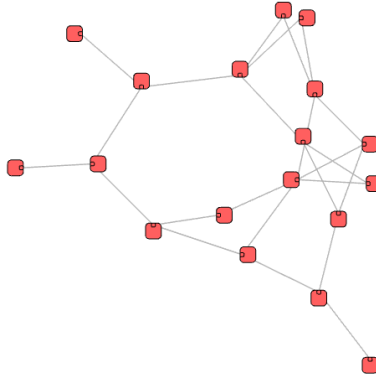


FIGURE A.6: Graph in PORGY

To get the graph with the attributes discussed in Chapter 4 we use rewrite rules and strategies and a simple Python script.

A.2.1 Visual rewrite rules and strategies

We will need to make rules and strategies to modify:

- edges of type CP_r and OG :
- nodes of every type except D .

We will now give examples of such rules and strategies.

Example A.1. We begin with an example on modifying edges of type CP_r . These edges will be turned green if the value of their *auth* attribute is *A* or red if it is *B*. In Figure A.7 we show one of these rules, composed of two nodes and one edge in both sides, where:

- L_W is defined by $N_L = \{n_1, n_2\}$, $P_L = \{p_1, p_2\}$, $E_L = \{e_1\}$, $W = \emptyset$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, C)\}$;
 - $\mathcal{L}_L(n_2) = \{(Interface, \{p_2\}), (ent, v_2), (type, P_r)\}$;
 - $\forall p_i \in N_L, \mathcal{L}_L(p_i) = \{(Attach, n_i), (Arity, 1)\}$;
 - $\mathcal{L}_L(e_1) = \{(Connect, \{p_1, p_2\}), (type, CP_r), (auth, A), (viewColor, grey)\}$.
- R_M^N is defined by $N_R = \{n_1, n_2\}$, $P_R = \{p_1, p_2\}$, $E_R = \{e_1\}$, $M = R$, $N = \emptyset$ and:
 - $\forall n_i \in N_L, \mathcal{L}_R(n_i) = \mathcal{L}_L(n_i)$;
 - $\forall p_i \in P_L, \mathcal{L}_R(p_i) = \mathcal{L}_L(p_i)$;
 - $\mathcal{L}_R(e_1) = \{(Connect, \{p_1, p_2\}), (type, CP_r), (auth, A), (viewColor, green)\}$.

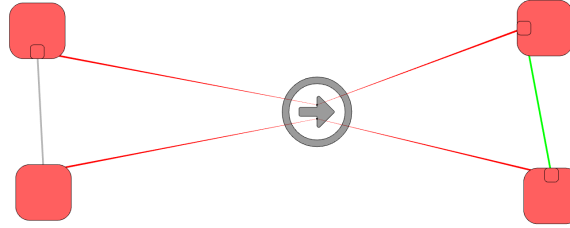


FIGURE A.7: Rewrite rule visualCPra

Note that the value of the *viewColor* attribute is set to grey on the left edge and green on the right edge, this ensures that the rule will only apply to each edge of this type one time. To use this rule we simply write the strategy:

```
repeat(one(visualCPra))
```

Now let's focus on nodes. Rewrite rules to modify nodes will be composed of one node on both sides. One such rule is shown in Figure A.8, where:

- L_W is defined by $N_L = \{n_1\}$, $P_L = \{p_1\}$, $E_L = \emptyset$, $W = \emptyset$ and:
 - $\mathcal{L}_L(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, "")\}$;
 - $\mathcal{L}_L(p_1) = \{(Attach, n_1), (Arity, 0)\}$.

- R_M^N is defined by $N_R = \{n_1\}$, $P_R = \{p_1\}$, $E_R = \emptyset$, $M = R$, $N = \emptyset$ and:
 - $\mathcal{L}_R(n_1) = \{(Interface, \{p_1\}), (ent, v_1), (type, P), (viewLabel, v_1), (viewColor, yellow), (viewShape, Pentagon)\}$;
 - $\mathcal{L}_R(p_1) = \{(Attach, n_1), (Arity, 0), (viewColor, white)\}$.

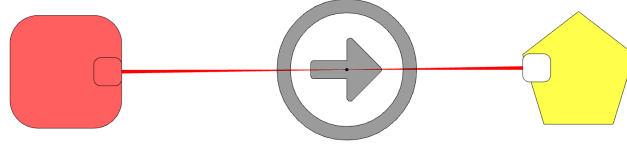


FIGURE A.8: Rewrite rule visualPp

Note that the value of the *viewLabel* attribute is set to "" on the left node and v_1 on the right node, this ensures that the rule will apply to each node of this type only once. Nodes of type *O* and *G* will only be blue and, as such, they will only need one rule of this kind for each one and the strategy to use it resembles the one for edges. Nodes of type *E* can be blue or light blue (one of them) so they need two rules but the strategies to apply them also resemble the one for edges. Nodes of type *P*, *C*, P_r , *A* and *R* on the other hand might be yellow, green or blue. So for each type of these nodes we will have three rules that only differ in the color of the right node and the strategies to apply them are little more complex. The following is strategy *visualP* that applies the rules to modify nodes of type *P*:

```

setBan(all(
  ngb(property(crtGraph, edge, type=="C0"), edge, type=="PC") [cup]
  ngb(property(crtGraph, edge, type=="CC" && obl=="T"), edge, type=="PC") [cup]
  property(crtGraph, edge, type=="EP") [cup]
  property(crtGraph, edge, type=="DP")
));
repeat(one(visualPp));
setBan(all(
  ngb(property(crtGraph, edge, type=="CMN"), edge, type=="PC") [cup]
  ngb(property(crtGraph, edge, type=="CC" && auth=="T"), edge, type=="PC")
));
repeat(one(visualPo));
setBan(all([emptySet]));
repeat(one(visualPpo))

```

To understand this strategy it is worth noting that the lowercase "p" and "o" in the name of rules stand for permission and obligation, respectively. Rule *visualPp* is supposed to be used to modify nodes that are only in the permission subgraph (yellow color), rule *visualPo* for nodes only in the obligation subgraph (blue color) and *visualPpo* for nodes in both subgraphs (green color). With this in mind the strategy works like this:

1. we put in the banned subgraph every node of type P that has any relation (edge) with the obligation subgraph;
2. we use rule *visualPp* to modify the remaining nodes of type P (the ones who do not have any relation with the obligation subgraph);
3. we put in the banned subgraph every node of type P that has any relation (edge) with the permission subgraph;
4. we use rule *visualPo* to modify the remaining nodes of type P (the ones who do not have any relation with the permission subgraph);
5. we set the banned subgraph as empty;
6. the rule *visualPpo* modifies the nodes of type P not yet modified (the ones that have relations with both subgraphs).

The strategies for the other nodes follow the same scheme. To apply the rules for every edge and node we have strategy *visual*:

```
repeat(one(visualCPra));repeat(one(visualCPrb));
repeat(one(visualOGi));repeat(one(visualOGf));
#visualP#;#visualC#;#visualPr#;#visualA#;#visualR#;
repeat(one(visualO));repeat(one(visualG));
repeat(one(visualEn));repeat(one(visualE))
```

Applying this strategy to the graph in Figure A.6 we get the graph shown in Figure A.9. Using this strategy we get the graph with almost all of the properties discussed in Chapter 4, the only thing that is missing is making the edges of type CC, GG and EE connect to the **In** and **Out** ports of the nodes they connect to. To do this we use a simple Python script.

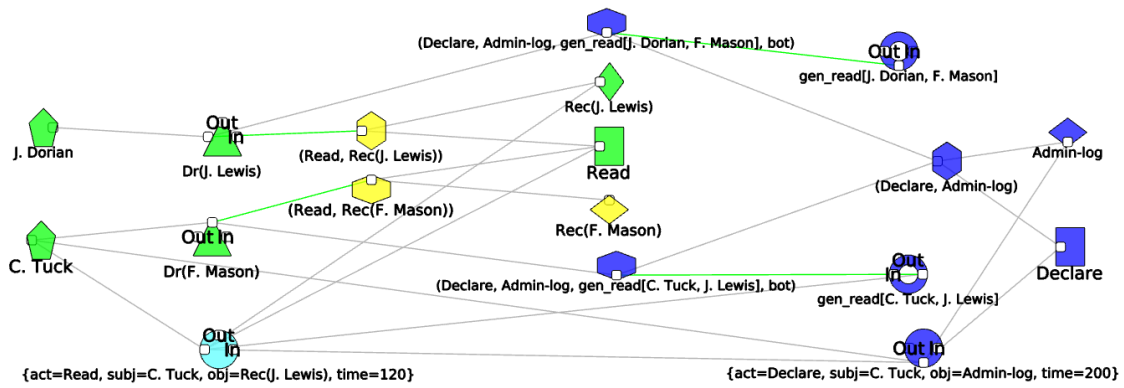


FIGURE A.9: Graph after applying strategy visual

A.2.2 Python script

If you look closely at the graph in Figure A.9 you can see that the edge of *type EE* is connected to the same port as all the other edges, but it should connect to the **Out** port of the left node and to the **In** port of the right node. To change this we use the following Python script:

```
from tulip import tlp

def main(graph):

    ent = graph['ent']
    target = graph['target']
    type_ = graph['type']
    viewLabel = graph['viewLabel']

    for n in graph.getNodes():
        if type_[n] in ["C", "E", "G"]:
            for n1 in graph.getOutNodes(n):
                if viewLabel[n1] == "":
                    regPort = n1
                elif viewLabel[n1] == "Out":
                    outPort = n1
                elif viewLabel[n1] == "In":
                    inPort = n1
            for e in graph.getInOutEdges(regPort):
                if type_[e] in ["CC", "EE", "GG"]:
                    if ent[n] in target[e]:
                        if graph.target(e) == regPort:
                            graph.setTarget(e, inPort)
                        else:
                            graph.setSource(e, inPort)
                    else:
```

```

if graph.target(e) == regPort:
    graph.setTarget(e, outPort)
else:
    graph.setSource(e, outPort)

```

This script receives a graph which you can chose in the Tulip Python IDE of PORGY. To understand the script it is worth noting that ports are represented as nodes that are connected by an edge to the actual node and, knowing this:

- `graph['attribute name']` returns some sort of dictionary where you can use an element of the graph as key to get the value of said attribute;
- `graph.getNodes()` returns an iterable list of all the nodes of the graph;
- `graph.getOutNodes(n)` returns an iterable list of the nodes connected to `n`;
- `graph.getInOutEdges(n)` returns an iterable list of the edges that connect to `n`;
- `graph.target(e)` (resp. `graph.source(e)`) returns the target (resp. source) node that edge `e` connects to;
- `graph.setTarget(e,n)` (resp. `graph.setSource(e,n)`) sets the target (resp. source) of edge `e` to be `n`.

The script works by going through all the nodes in the graph. When it finds a node `n` of type *C*, *E* or *G*:

1. it saves the nodes connected to it (which are its ports) as `regPort`, `outPort` or `inPort`;
2. it goes through all the edges connected to `regPort`. When it finds an edge of type *CC*, *EE* or *GG*:
 - (a) if the value of attribute `ent` of the node `n` is in the value of attribute `target` of the edge, it connects the edge to the `inPort`;
 - (b) otherwise, it connects the edge to the `outPort`.

Using this script in the Tulip Python IDE of PORGY, shown in Figure [A.10](#), and making sure to select the right graph (G37 in our case) gives us the graph shown in Figure [A.11](#), where the edge of type *EE* is now connected to the right ports.

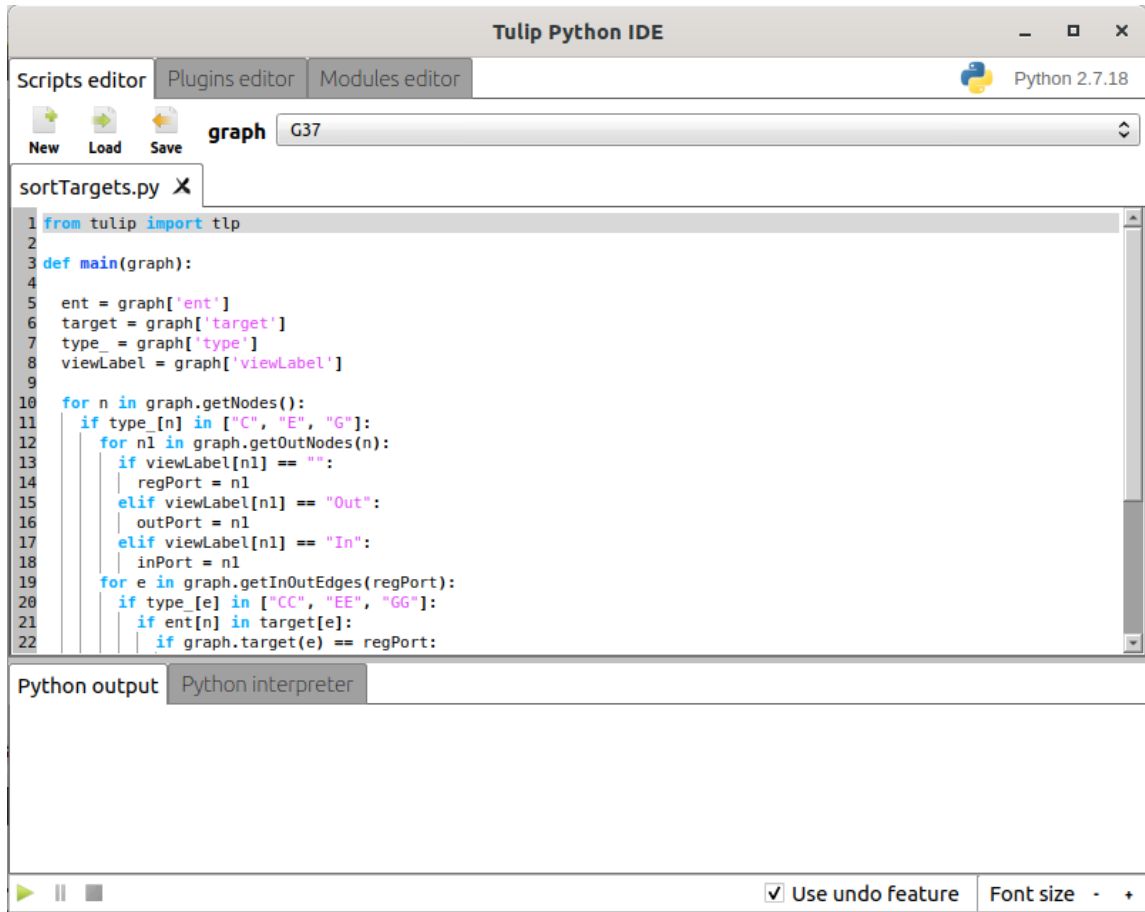


FIGURE A.10: Tulip Python IDE in PORGY

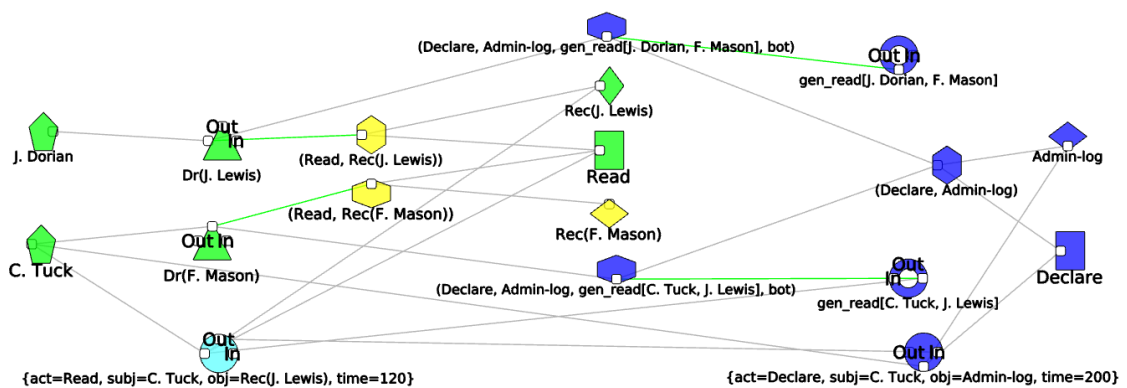


FIGURE A.11: Graph after using Porgy script

Appendix B

Value variables in PORGY

Value variables are used in every rewrite rule in this work, they are the values v_i seen in the records of elements of the rules. PORGY does not have a way of "writing" such variables in the record of elements. Instead, we can use the Algorithm configuration tab when creating a rule to set how the values of attributes of elements of L are computed into R . In Figure B.1 we see the rule `visualPp` shown and defined in the previous appendix with its algorithm.

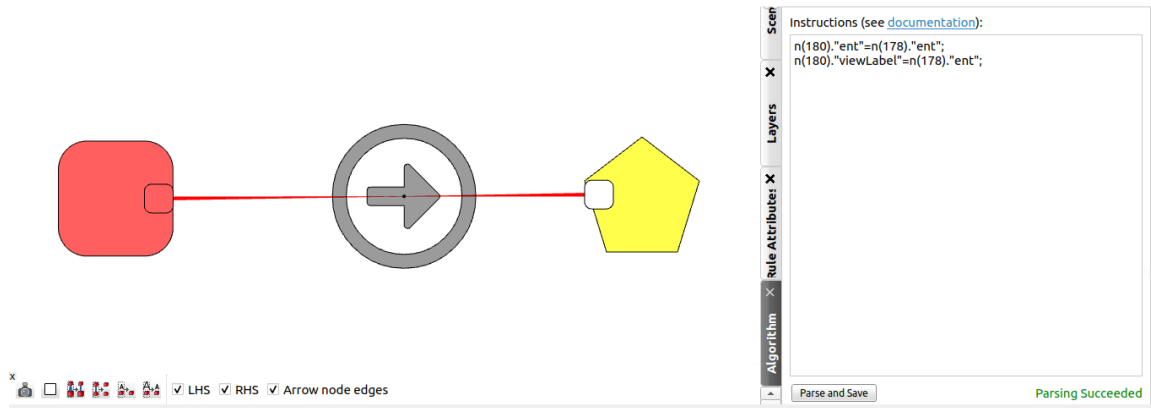


FIGURE B.1: Algorithm tab in PORGY

In this example we see the value of the *ent* attribute of the left node, `n(178)`, being copied into the values of attributes *ent* and *viewLabel* of the right node, `n(180)`. Concatenation of strings can also be used using the `+` sign and values of edges and ports can also be used. For more information we refer the reader to [14].

Bibliography

- [1] S. Barker, “The next 700 access control models or a unifying meta-model?” in *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 187–196. [Online]. Available: <https://doi.org/10.1145/1542207.1542238> [Cited on pages 1 and 95.]
- [2] S. Alves, A. Degtyarev, and M. Fernández, “Access control and obligations in the category-based metamodel: A rewrite-based semantics,” in *Logic-Based Program Synthesis and Transformation*, M. Proietti and H. Seki, Eds. Springer, 2015, pp. 148–163. [Cited on pages 2, 9, and 100.]
- [3] S. Alves and M. Fernández, “A graph-based framework for the analysis of access control policies,” *Theoretical Computer Science*, vol. 685, pp. 3 – 22, 2017, logical and Semantic Frameworks with Applications. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0304397516305965> [Cited on pages 2, 11, and 95.]
- [4] O. Andrei, M. Fernández, H. Kirchner, G. Melançon, O. Namet, and B. Pinaud, “PORGY: Strategy-Driven Interactive Transformation of Graphs,” in *TERMGRAPH*, 2011, pp. 54–68. [Cited on pages 2, 16, and 101.]
- [5] O. Andrei, “A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems,” Theses, Institut National Polytechnique de Lorraine - INPL, Nov. 2008. [Online]. Available: <https://tel.archives-ouvertes.fr/tel-00337558> [Cited on pages 2 and 16.]
- [6] D. Auber, D. Archambault, R. Bourqui, M. Delest, J. Dubois, A. Lambert, P. Mary, M. Mathiaut, G. Melançon, B. Pinaud, B. Renoust, and J. Vallet, *Tulip*

5. New York, NY: Springer New York, 2017, pp. 1–28. [Online]. Available: https://doi.org/10.1007/978-1-4614-7163-9_315-1 [Cited on pages 2 and 101.]
- [7] S. Alves and J. Iglésias, “A graphical framework for the category-based metamodel for access control and obligations,” *CoRR*, vol. abs/2111.00588, 2021. [Online]. Available: <https://arxiv.org/abs/2111.00588> [Cited on page 2.]
- [8] F. Harary, *Graph Theory*. Reading, MA: Addison-Wesley, 1969. [Cited on page 3.]
- [9] C. Bertolissi and M. Fernández, “Category-based authorisation models: Operational semantics and expressive power,” in *Engineering Secure Software and Systems*, F. Mas-sacci, D. Wallach, and N. Zannone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 140–156. [Cited on pages 4 and 95.]
- [10] F. Baader and T. Nipkow, *Term Rewriting and All That*. USA: Cambridge University Press, 1999. [Cited on page 4.]
- [11] S. Alves, S. Broda, and M. Fernández, “A typed language for events,” in *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Falaschi, Ed., vol. 9527. Springer, 2015, pp. 107–123. [Online]. Available: https://doi.org/10.1007/978-3-319-27436-2_7 [Cited on page 7.]
- [12] S. Alves and M. Fernández, “A framework for the analysis of access control policies with emergency management,” *Electronic Notes in Theoretical Computer Science*, vol. 312, pp. 89 – 105, 2015, ninth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1571066115000109> [Cited on pages 11 and 95.]
- [13] M. Fernández, H. Kirchner, and B. Pinaud, “Strategic Port Graph Rewriting: An Interactive Modelling and Analysis Framework,” Inria, Research Report, Jan. 2016. [Online]. Available: <https://hal.inria.fr/hal-01251871> [Cited on page 16.]
- [14] —, “Strategic Port Graph Rewriting: an Interactive Modelling Framework,” *Mathematical Structures in Computer Science*, vol. 29, no. 5, pp. 615–662, May 2019. [Online]. Available: <https://hal.inria.fr/hal-01251871> [Cited on pages 16 and 113.]

- [15] O. Andrei and H. Kirchner, "A Rewriting Calculus for Multigraphs with Ports," in *The Eighth International Workshop on Rule-Based Programming (RULE 2007)*. Paris, France: ENTCS, Jun. 2007, Technical Report, p. 20. [Online]. Available: <https://hal.inria.fr/inria-00139363> [Cited on page 16.]
- [16] C. Bertolissi and M. Fernández, "Rewrite specifications of access control policies in distributed environments," in *Security and Trust Management*, J. Cuellar, J. Lopez, G. Barthe, and A. Pretschner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 51–67. [Cited on page 95.]
- [17] C. Bertolissi and M. Fernández, "A metamodel of access control for distributed environments: Applications and properties," *Information and Computation*, vol. 238, pp. 187 – 207, 2014, special Issue on Security and Rewriting Techniques. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0890540114000959>
- [18] A. Ali and M. Fernández, "Hybrid enforcement of category-based access control," in *Security and Trust Management*, S. Mauw and C. D. Jensen, Eds. Cham: Springer International Publishing, 2014, pp. 178–182. [Cited on page 95.]
- [19] H. Mirzapour-Aghdaghi and M. Fernández. (2014) Policy manager: a tool to analyse category-based access control policies. [Online]. Available: <http://policymanager.herokuapp.com> [Cited on page 95.]
- [20] J. Sá, S. Alves, and S. Broda, "The g-acm tool: using the drools rule engine for access control management," 2016. [Cited on page 95.]
- [21] R. W. Baldwin, "Naming and grouping privileges to simplify security management in large databases," in *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, 1990, pp. 116–132. [Cited on page 96.]
- [22] M. Nyanchama and S. Osborn, "The role graph model and conflict of interest," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 1, p. 3–33, Feb. 1999. [Online]. Available: <https://doi.org/10.1145/300830.300832> [Cited on page 96.]
- [23] M. Koch, L. V. Mancini, and F. Parisi-Presicce, "A formal model for role-based access control using graph transformation," in *Computer Security - ESORICS 2000*, F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 122–139. [Cited on page 96.]

- [24] M. Koch, L. Mancini, and F. Parisi-Presicce, "A graph-based formalism for rbac," *ACM Transactions on Information and System Security*, vol. 5, no. 3, pp. 332–365, 2002, cited By 100. [Online]. Available: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85008430937&doi=10.1145%2f545186.545191&partnerID=40&md5=7565d762d7e19aef15f62f985d797f06> [Cited on page 96.]
- [25] R. Thion and S. Coulondre, "Representation and reasoning on role-based access control policies with conceptual graphs," in *Conceptual Structures: Inspiration and Application*, H. Schärfe, P. Hitzler, and P. Øhrstrøm, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 427–440. [Cited on pages 96 and 97.]
- [26] M. Koch, L. V. Mancini, and F. Parisi-Presicce, "On the specification and evolution of access control policies," in *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 121–130. [Online]. Available: <https://doi.org/10.1145/373256.373280> [Cited on page 96.]
- [27] —, "Foundations for a graph-based approach to the specification of access control policies," in *Foundations of Software Science and Computation Structures*, F. Honsell and M. Miculan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 287–302. [Cited on page 96.]
- [28] M. Koch and F. Parisi-Presicce, "Visual specifications of policies and their verification," in *Fundamental Approaches to Software Engineering*, M. Pezzè, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 278–293. [Cited on page 96.]
- [29] J. A. Hoagland, "Specifying and implementing security policies using lasco, the language for security constraints on objects," *CoRR*, vol. cs.CR/0003066, 2000. [Online]. Available: <https://arxiv.org/abs/cs/0003066> [Cited on page 97.]
- [30] A. Heydon, M. Maimone, J. Tygar, J. Wing, and A. Zaremski, "Miro: Visual specification of security," *Software Engineering, IEEE Transactions on*, vol. 16, pp. 1185 – 1197, 11 1990. [Cited on page 97.]
- [31] D. Harel, "On visual formalisms," *Commun. ACM*, vol. 31, no. 5, p. 514–530, May 1988. [Online]. Available: <https://doi.org/10.1145/42411.42414> [Cited on page 97.]

- [32] J. E. Tidswell and J. M. Potter, "A graphical definition of authorization schema in the dtac model," in *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 109–120. [Online]. Available: <https://doi.org/10.1145/373256.373276> [Cited on page 98.]
- [33] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995. [Cited on page 101.]