EPTCS 410

Proceedings of the

Eighth Symposium on Working Formal Methods

Timişoara, Romania, September 16-18

Edited by: Mircea Marin and Laurenţiu Leuştean

Published: 31st October 2024
DOI: 10.4204/EPTCS.410
ISSN: 2075-2180
Open Publishing Association

Preface

Mircea Marin
West University of Timişoara
Timişoara, Romania
mircea.marin@e-uvt.ro

Laurenţiu Leuştean
University of Bucharest & ILDS & IMAR
Bucharest, Romania
laurentiu.leustean@unibuc.ro

The Working Formal Methods Symposium (FROM) is a series of workshops that aim to bring together researchers and practitioners who work on formal methods by contributing new theoretical results, methods, techniques, and frameworks, and/or by creating or using software tools that apply theoretical contributions.

This volume contains the papers presented at the Eighth Working Formal Methods Symposium (FROM 2024) held in Timişoara, Romania on September 16-18, 2024. The symposium was organised by the West University of Timişoara and was co-located with the Symbolic and Numaric Algorithms for Scientific Computing (SYNASC 2024) conference.

The scientific program consisted of two invited talks by:

- Laura Kovacs (TU Wien, Austria)
- Nikolaj Bjorner (Microsoft Research, USA)

and four tutorials by

- Vlad Rusu (Inria Lille, France)
- Andrei Voronkov (University of Manchester, England)
- Isabela Drămnesc (West University of Timișoara, Romania)
- Mădălina Erașcu (West University of Timișoara, Romania)

and eleven contributed papers. The members of the Programme Committee for the workshop were:

- Florin Crăciun (Babeş-Bolyai University of Cluj-Napoca)
- Volker Diekert (Universität Stuttgart)
- Radu Iosif (Verimag, CNRS, University of Grenoble Alpes)
- Temur Kutsia (Johannes Kepler University Linz)
- Laurenţiu Leuştean (University of Bucharest & ILDS & IMAR) (co-chair)
- Dorel Lucanu (Alexandru Ioan Cuza University of Iaşi)
- Mircea Marin (West University of Timişoara) (co-chair)
- David Nowak (CNRS & University of Lille)
- Corina Păsăreanu (NASA & Carnegie Mellon University)
- Thomas Powell (University of Bath)
- Grigore Roşu (University of Illinois at Urbana-Champaign)
- Vlad Rusu (INRIA Lille)
- Andrei Sipoş (University of Bucharest & ILDS & IMAR)

- Viorica Sofronie-Stokkermans (University of Koblenz and Landau)
- Alicia Villanueva (VRAIN – Universitat Politècnica de València)

We would like to thank the members of the programme committee and the reviewers for their effort, the authors for their contributions, EPTCS for publishing this volume and Rob van Glabbeek for his immense support to us as editors. We are also grateful for the generous support of our sponsors: ILDS - Institute for Logic and Data Science, BRD – Groupe Société Générale, and Runtime Verification.

Mircea Marin
Laurenţiu Leuştean
Editors

Timişoara, 2024

Table of Contents

Unification in Matching Logic — Revisited

Ádám Kurucz
ELTE Eötvös Loránd University
Budapest, Hungary
cphfw1@inf.elte.hu

Péter Bereczky
ELTE Eötvös Loránd University
Budapest, Hungary
berpeti@inf.elte.hu

Dániel Horpácsi
ELTE Eötvös Loránd University
Budapest, Hungary
daniel-h@elte.hu

Matching logic is a logical framework for specifying and reasoning about programs using pattern matching semantics. A pattern is made up of a number of structural components and constraints. Structural components are syntactically matched, while constraints need to be satisfied. Having multiple structural patterns poses a practical problem as it requires multiple matching operations. This is easily remedied by unification, for which an algorithm has already been defined and proven correct in a sorted, polyadic variant of matching logic. This paper revisits the subject in the applicative variant of the language while generalising the unification problem and mechanizing a proven-sound solution in Coq.

1 Introduction

First-order unification is the process of solving equations between first-order terms (symbolic expressions). In particular, unification of first-order terms t_1 and t_2 yields a substitution σ such that it makes the two terms syntactically equal (identical): $t_1\sigma = t_2\sigma$. Unification plays a crucial role in automatic theorem proving via first-order resolution as well as in term rewriting.

Matching logic is a general logical framework for specifying and reasoning about programming languages and programs using pattern matching semantics. In matching logic, formulas (called patterns) are evaluated to a subset of a domain, and those evaluating to the full set are valid. Matching logic is the formal meta-language of the K framework [5], hosting programming language semantics in terms of constrained rewrite systems. Formal reasoning in these theories requires a sound unification algorithm supporting, amongst others, the reduction of patterns specifying program states.

Despite the fact that matching logic is not an equational logic, syntactic unification has a semantic counterpart in it, based on the notion of equality derived from other, lower-level operations. Given two matching logic patterns φ_1 and φ_2, their semantic unifier is a pattern φ such that $\varphi \to \varphi_1 = \varphi_2$, where "$=$" denotes semantic equality rather than syntactic equality.

In [1] Arusoaie et al. show that in a sorted, polyadic variant of matching logic [9], semantic unification can be derived from syntactic unification: tailoring a well-known rule-based unification algorithm, they obtain the most general unifier substitution σ, from which they trivially construct a semantic unifier ϕ^σ and then they show that for patterns t_1 and t_2, $t_1\sigma = t_2\sigma$ implies $\phi^\sigma \leftrightarrow t_1 = t_2$, allowing the equality pattern to be replaced by the unification pattern due to the congruence rule admitted by matching logic.

Following the footsteps of Arusoaie et al. we adapt the syntactic unification algorithm to the applicative, unsorted variant of matching logic and derive the semantic unifier from the solution. Similarly to their work, we assume that the theory specifies a term algebra and the patterns represent constrained first-order terms (i.e., symbols are injective constructor functions). Furthermore, derived from the soundness of the semantic unification, we show that $t_1 \wedge t_2 = t_1 \wedge \phi^\sigma$, which allows multiple term patterns to be merged into a constrained term pattern, enabling more effective pattern matching and utilization of SMT solvers. The latter is based on the fact that in matching logic, $t_1 \wedge t_2$ is equivalent to $t_1 \wedge t_1 = t_2$, that is, the conjunction of two term patterns can be replaced by one of the term patterns and the predicate stating their semantic equality—such patterns are common in automatic theorem proving in the K framework.

However, compared to related work, the novelty of our approach lies in the fact that we prove the soundness of the unification on a fully syntactical basis, using a sound sequent calculus for matching logic [11]. Consequently, unlike in the work of Arusoaie et al., we do not need to synthesize proofs for particular unifications as our constructive soundness proof justifies the derivability of the equivalence between the unified patterns and the semantic unifier.

In this work we make the following contributions:

- The definition of a generalised, abstract unification problem;
- A rule-based unification algorithm for the unsorted, applicative variant of matching logic;
- Proof of the soundness of the unification algorithm, using a single-conclusion sequent calculus;
- Machine-checked implementation of the above results.

The rest of the paper is structured as follows. In Section 2 we introduce matching logic and the unification problem for first-order terms. Then in Section 3 we discuss existing solutions to solving unification and the related work for matching logic in particular. Thereafter, Section 4 defines abstract unification problems, a rule-based solution, and we present the proof of the soundness of the solution. Finally, Section 5 summarises the results and concludes.

2 Background

In this section, we provide a general introduction to matching logic [6] and we also recall some theorems and meta-theorems of matching logic that we will build upon when proving properties of unifier patterns. At the end of this section, we overview the unification problem for first-order terms in general.

2.1 Matching Logic

This section introduces an applicative variant of matching logic [6, 7] which is the language hosting the unification problem explained in this paper.

2.1.1 Syntax

Matching logic is a minimal yet expressive language. Its syntax is parametric in the so-called signature, containing constant symbols and variables.

Definition 1 (Signature)**.** In a simplified setting, a matching logic signature is a pair (EV, Σ), where

- EV is a countably infinite set of element variables (denoted with $x, y, \ldots$);
- Σ is a countable set of constant symbols (denoted with $f, g, \ldots$).

Whenever EV is understood from the context, only Σ is used to denote the whole signature.

The only syntactic category of the language is *patterns*, parametrised by a particular signature.

Definition 2 (Pattern)**.** Given a signature (EV, Σ), the following rules define patterns:

$$\varphi ::= x \mid f \mid \varphi_1\,\varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\, \varphi$$

These constructs are in order: (element) variables, symbols, application of patterns, falsum (bottom) and the traditional first-order operations (implication and existential quantification). Application is left associative and binds the tightest. Implication is right associative. The scope of the $\exists$ binder extends as far to the right as possible. We use $FV(\varphi)$ to refer to the set of free variables of some pattern φ.

Note that the full version of matching logic also contains least-fixpoint (μ) patterns [6], but they are not relevant for this work, thus for the sake of readability we omit them from the current presentation. The reason for this is that most of the patterns presented here are term patterns, which do not contain fixpoints syntactically. We note that the formalisation [8] includes μ-patterns too, but some mechanised theorems and rules (e.g., $(=\vdash)$) are restricted to work with μ-free patterns currently.

Matching logic is intentionally minimal, but can derive several other, well-known constructs as syntactic sugar. For instance, the following lines define negation, disjunction, conjunction, top, equivalence, and universal quantification.

$$\begin{aligned}
\neg\varphi &\overset{\text{def}}{=} \varphi \to \bot & \qquad \varphi_1 \vee \varphi_2 &\overset{\text{def}}{=} \neg\varphi_1 \to \varphi_2 \\
\varphi_1 \wedge \varphi_2 &\overset{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2) & \top &\overset{\text{def}}{=} \neg\bot \\
\varphi_1 \leftrightarrow \varphi_2 &\overset{\text{def}}{=} (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1) & \forall x.\, \varphi &\overset{\text{def}}{=} \neg\exists x.\, \neg\varphi
\end{aligned}$$

We define substitutions on patterns in the usual way.

Definition 3 (Substitution)**.** We denote substitutions as $\varphi[\psi/x]$ to say that we replace every free occurrence of x in φ with ψ.

Special patterns are pattern contexts and application contexts, which are always used in substitutions.

Definition 4 (Pattern context, application context)**.** A pattern context, denoted with C is a pattern with a distinguished variable $\square$, called a hole. When substituting φ into a hole, we simply write $C[\varphi]$ denoting $C[\varphi/\square]$. When a context contains only applications from the root to the (only) $\square$, we call it an application context. Application contexts are defined with the following grammar:

$$C^{\$} := \square \mid \varphi\, C^{\$} \mid C^{\$}\, \varphi$$

Contexts are useful for defining theorems that are only concerned with a smaller part of a pattern while the rest is irrelevant. Contexts are used for example to substitute equal patterns inside a bigger context (see rule $(=\vdash)$). This definition of contexts is slightly different from the definition used in [6, 7], but the same proofs can be expressed with both variants[1].

2.1.2 Semantics

Matching logic semantics is based on pattern matching. Patterns are interpreted as a set of domain elements that match them.

Definition 5 (Model)**.** A model is a tuple $(M, \_\cdot\_, M_f)$, where

- M is the non-empty carrier set (domain), which contains all elements a pattern may evaluate to,
- $\_\cdot\_ : M \to M \to \mathscr{P}(M)$ is a binary function that serves as the interpretation of application,
- $M_f \subseteq M$ is the (indexed) interpretation of symbols $f \in \Sigma$.

[1]This correspondence is proven in the Coq formalisation [8, `matching-logic/src/ProofMode/Misc.v`].

We extend the application function to subsets of the domain in a pointwise manner:

$$A \cdot B = \bigcup_{\substack{a \in A \\ b \in B}} a \cdot b$$

Let us define the semantics of matching logic formulas informally. Given a signature and a model, we can define interpretations for patterns. Element variables are interpreted as singleton sets, while the meaning of symbols is given by M_f. In the case of application, the meanings of the subpatterns are combined using the $\_\cdot\_$ function. $\bot$ is matched by the empty set and the implication $\varphi_1 \to \varphi_2$ is matched by elements that match φ_2 if they match φ_1. $\exists x.\, \varphi$ is matched by all instances of φ as x ranges over M. This means, for example, that $\exists x. x$ is matched by M. The semantics of the derived operations follow from these.

Definition 6 (Semantic consequence)**.** A pattern is validated by a model if the pattern evaluates to the entire carrier set of the model. A model validates a set of patterns, if it validates all patterns in the set. In particular, we use $\Gamma \vDash \varphi$ to denote that the patterns in Γ (i.e., the set of axioms) validate a pattern φ whenever all models validating Γ also validate φ.

Definition 7 (Term patterns and predicate patterns)**.** We distinguish two classes of patterns that will be important in the later sections of this paper. Specifically, *term patterns* evaluate to singleton sets (they behave like terms in first-order logic), and *predicate patterns* evaluate either to the empty set or to the full carrier set (they mimic formulas in first-order logic).

2.1.3 Sequent Calculus for Matching Logic

Matching logic has a sound Hilbert-style proof system [6] with a handful of simple rules, and a sequent calculus [11] which is sound and complete w.r.t. the proof system. In this section, we briefly summarize this calculus, and we construct our proofs with it in later sections. We denote provability in the Hilbert-system with $\Gamma \vdash \varphi$, and next we define sequents.

Definition 8 (Sequents)**.** A *sequent* is a triple $\Gamma \blacktriangleright \Delta \vdash_S \psi$ [2], where

- Γ is a (possibly infinite) set of patterns, called a *theory*;
- Δ is a finite (comma-separated) list of patterns, called *antecedent* or *local context*;
- ψ is a pattern, called *succedent* or *conclusion*.

Next, we recall the sequent calculus in Figure 1 and refer to [11] for further details. Essentially, these rules mimic a standard single-conclusion sequent calculus for first-order logic, except that they use a concept of *local context* and do not affect the theory (Γ) directly. Informally, the meaning of the sequent $\Gamma \blacktriangleright \varphi_1, \ldots, \varphi_n \vdash_S \psi$ can be expressed as an expansion to $\Gamma \vdash \varphi_1 \to \cdots \to \varphi_n \to \psi$, which allows the premises to be separated from the conclusion without using a deduction theorem. With this notion in mind, we can see that the rule ($\vdash\!\to$) is not a deduction theorem of matching logic (we refer to Section 2.1.4 for further details), but in practice it functions as a rule turning implication premises to hypotheses. The sequent calculus accommodates another rule for the deduction meta-theorem (see DEDUCTION), but in the general case (when the fixed-point operator is present in the language) that comes with technical side conditions [11], making it more difficult to apply.

[2]Note that $\vdash_S$ denotes a derivation in the sequent calculus, whereas $\vdash$ denotes a derivation in the Hilbert-style proof system.

$$\frac{\Gamma \vdash \psi}{\Gamma \blacktriangleright [] \vdash_S \psi}\ \textsc{Inherit} \qquad \frac{\Gamma \blacktriangleright \Delta_1, \Delta_2 \vdash_S \psi}{\Gamma \blacktriangleright \Delta_1, \varphi, \Delta_2 \vdash_S \psi}\ \textsc{Weaken} \qquad \frac{\Gamma \blacktriangleright \Delta_1 \vdash_S \varphi \quad \Gamma \blacktriangleright \Delta_1, \varphi, \Delta_2 \vdash_S \psi}{\Gamma \blacktriangleright \Delta_1, \Delta_2 \vdash_S \psi}\ \textsc{Cut}$$

(a) Technical and structural inference rules

$$\frac{}{\Gamma \blacktriangleright \Delta_1, \varphi, \Delta_2 \vdash_S \varphi}\ \textsc{Hyp} \qquad \frac{}{\Gamma \blacktriangleright \Delta_1, \bot, \Delta_2 \vdash_S \psi}\ (\vdash \bot)$$

$$\frac{\Gamma \blacktriangleright \Delta_1, \Delta_2 \vdash_S \varphi_1 \quad \Gamma \blacktriangleright \Delta_1, \varphi_2, \Delta_2 \vdash_S \psi}{\Gamma \blacktriangleright \Delta_1, \varphi_1 \to \varphi_2, \Delta_2 \vdash_S \psi}\ (\to\vdash) \qquad \frac{\Gamma \blacktriangleright \Delta, \varphi \vdash_S \psi}{\Gamma \blacktriangleright \Delta \vdash_S \varphi \to \psi}\ (\vdash\to)$$

$$\frac{\Gamma \blacktriangleright \Delta_1, \varphi_1, \varphi_2, \Delta_2 \vdash_S \psi}{\Gamma \blacktriangleright \Delta_1, \varphi_1 \wedge \varphi_2, \Delta_2 \vdash_S \psi}\ (\wedge \vdash) \qquad \frac{\Gamma \blacktriangleright \Delta \vdash_S \psi_1 \quad \Gamma \blacktriangleright \Delta \vdash_S \psi_2}{\Gamma \blacktriangleright \Delta \vdash_S \psi_1 \wedge \psi_2}\ (\vdash \wedge)$$

$$\frac{\Gamma \blacktriangleright \Delta \vdash_S \psi_1}{\Gamma \blacktriangleright \Delta \vdash_S \psi_1 \vee \psi_2}\ (\vdash \vee_L)$$

$$\frac{\Gamma \blacktriangleright \Delta_1, \varphi_1, \Delta_2 \vdash_S \psi \quad \Gamma \blacktriangleright \Delta_1, \varphi_2, \Delta_2 \vdash_S \psi}{\Gamma \blacktriangleright \Delta_1, \varphi_1 \vee \varphi_2, \Delta_2 \vdash_S \psi}\ (\vee \vdash)$$

$$\frac{\Gamma \blacktriangleright \Delta \vdash_S \psi_2}{\Gamma \blacktriangleright \Delta \vdash_S \psi_1 \vee \psi_2}\ (\vdash \vee_R)$$

(b) Inference rules for propositional reasoning

$$\frac{\Gamma \blacktriangleright \Delta_1, \varphi[y/x], \Delta_2 \vdash_S \psi}{\Gamma \blacktriangleright \Delta_1, \forall x.\, \varphi, \Delta_2 \vdash_S \psi}\ (\forall \vdash) \qquad \frac{\Gamma \blacktriangleright \Delta \vdash_S \psi[y/x]}{\Gamma \blacktriangleright \Delta \vdash_S \forall x.\, \psi}\ y \notin FV(\Delta, \forall x.\, \psi) \quad (\vdash \forall)$$

$$\frac{\Gamma \blacktriangleright \Delta_1, \varphi[y/x], \Delta_2 \vdash_S \psi}{\Gamma \blacktriangleright \Delta_1, \exists x.\, \varphi, \Delta_2 \vdash_S \psi}\ y \notin FV(\Delta, \exists x.\, \varphi, \psi) \quad (\exists \vdash) \qquad \frac{\Gamma \blacktriangleright \Delta \vdash_S \psi[y/x]}{\Gamma \blacktriangleright \Delta \vdash_S \exists x.\, \psi}\ (\vdash \exists)$$

(c) Inference rules for first-order reasoning

Figure 1: Sequent calculus for matching logic (theory-independent rules)

We also highlight the rule INHERIT, which allows us to lift any proofs done with the Hilbert-style proof system ($\vdash$) into the sequent calculus. In later sections, we use this rule implicitly to lift Hilbert-style proofs of theorems while presenting a sequent calculus proof (note that with the repeated use of WEAKEN we can unify the local context of the lifted theorem and the current proof state). We also note that it is also possible to express all sequent calculus proof in the Hilbert-style proof system too (Theorem 1). For further insights on the correspondence between the calculus and the proof system, we refer to the paper on the calculus [11] and its implementation [8, `matching-logic/src/ProofMode`].

Next, we encode the theory of definedness (also featuring equality) in matching logic, and extend the sequent calculus with rules using definedness and equality, notably deduction and rewriting.

2.1.4 Definedness

We recall the formal specification for the theory of *definedness* from [6] in Figure 2. The section "Symbol" includes the new symbols added to the signature, while "Notation" describes the derived notations used in the theory. Finally, "Axiom" describes the axiom schemas that specify the symbols of the theory. In the rest of the paper, we will refer to the axiom set of the definedness theory with Γ^{DEF}.

spec DEFINEDNESS
Symbol: $\lceil \_ \rceil$
Notation:

$$\lceil \varphi \rceil \overset{\text{def}}{=} \lceil \_ \rceil\, \varphi \qquad \lfloor \varphi \rfloor \overset{\text{def}}{=} \neg \lceil \neg \varphi \rceil$$
$$\varphi_1 = \varphi_2 \overset{\text{def}}{=} \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor \qquad \varphi_1 \neq \varphi_2 \overset{\text{def}}{=} \neg(\varphi_1 = \varphi_2)$$
$$\varphi_1 \in \varphi_2 \overset{\text{def}}{=} \lceil \varphi_1 \wedge \varphi_2 \rceil \qquad \varphi_1 \notin \varphi_2 \overset{\text{def}}{=} \neg(\varphi_1 \in \varphi_2)$$
$$\varphi_1 \subseteq \varphi_2 \overset{\text{def}}{=} \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor \qquad \varphi_1 \not\subseteq \varphi_2 \overset{\text{def}}{=} \neg(\varphi_1 \subseteq \varphi_2)$$

Axiom:
(DEFINEDNESS) $\lceil x \rceil$
endspec

Figure 2: Specification of definedness

This theory introduces a symbol $\lceil \_ \rceil$, a number of notations on top of this symbol, and one axiom which embodies the meaning of the definedness symbol. The axiom states that all (element) variables are defined, i.e., the definedness applied to a variable is always satisfied. Intuitively, $\lceil \varphi \rceil$ is satisfied if φ matches at least one element. Based on definedness, we can derive *totality* (denoted as $\lfloor \varphi \rfloor$), and the usual notion of equality, membership and set inclusion. Intuitively, $\lfloor \varphi \rfloor$ is satisfied if φ matches all elements of the domain. With equality, we can syntactically express when a pattern is a term pattern (also called functional pattern), or a predicate pattern. We can also show when symbol f behaves like an n-ary function, or as an n-ary predicate:

$$\exists x.\, \varphi = x \qquad \text{(Functional Pattern)}$$
$$\varphi = \bot \vee \varphi = \top \qquad \text{(Predicate Pattern)}$$
$$\forall x_1.\dots\forall x_n.\,\exists y.\, f\, x_1 \cdots x_n = y \qquad \text{(Function)}$$
$$\forall x_1.\dots\forall x_n.\, g\, x_1 \cdots x_n = \top \vee g\, x_1 \cdots x_n = \bot \qquad \text{(Predicate)}$$

For reasoning about equality, we use the rules presented in Figure 3. All rules of the sequent calculus are proven sound in [11]. Note that both the deduction rule and the rewriting rule (replacing equal patterns) are based on the deduction theorem and the equality elimination theorem, which in their general form come with complex technical side conditions intentionally neglected in this presentation; for details about the technical side conditions, we refer to [7].

$$\frac{\Gamma \cup \{\varphi\} \blacktriangleright \Delta \vdash_S \psi\ ^{\ddagger}}{\Gamma \blacktriangleright \lfloor \varphi \rfloor, \Delta \vdash_S \psi}\ \text{DEDUCTION} \qquad \frac{}{\Gamma \blacktriangleright \Delta \vdash_S \psi = \psi}\ (\vdash=) \qquad \frac{\Gamma \blacktriangleright \Delta_1, \varphi_1 = \varphi_2, \Delta_2 \vdash_S C[\varphi_2]}{\Gamma \blacktriangleright \Delta_1, \varphi_1 = \varphi_2, \Delta_2 \vdash_S C[\varphi_1]}\ (=\vdash)$$

Figure 3: Deduction; and rules about equality. These rules assume that $\Gamma^{\mathsf{DEF}} \subseteq \Gamma$.

Next, we state important theorems and meta-theorems of matching logic which we will rely on when reasoning about the correctness of unification (we refer to [3, 6] for detailed explanations and proofs).

2.1.5 Essential Theorems

First of all, we state the correspondence theorem that allows us to use Hilbert-style proofs and sequent calculus proofs interchangeably. In particular, this allows us to reason about Hilbert-style provability by using the sequent calculus.

‡The proof does not use existential generalization on free variables of ψ.

Theorem 1 (Correspondence)**.** For all theories Γ, and patterns $\varphi_1, \dots, \varphi_k, \psi$, the derivability statement $\Gamma \blacktriangleright \varphi_1, \dots, \varphi_k \vdash_S \psi$ holds if and only if $\Gamma \vdash \varphi_1 \to \dots \to \varphi_k \to \psi$.

Now we state fundamental (meta-)theorems heavily used in the subsequent sections.

Theorem 2 (Congruence)**.** For all patterns φ_1, φ_2, pattern contexts C and theory Γ, if $\Gamma \vdash \varphi_1 \leftrightarrow \varphi_2$ then $\Gamma \vdash C[\varphi_1] \leftrightarrow C[\varphi_2]$.

Lemma 1. *Let Γ be a theory and φ a pattern. Then $\Gamma \vdash \varphi \to \lceil \varphi \rceil$.*

Note that Lemma 1 is a consequence of the DEFINEDNESS axiom and existential generalization.

Lemma 2. *Let Γ be a theory and φ_1 and φ_2 functional patterns. Then $\Gamma \vdash \varphi_1 \in \varphi_2 \to \varphi_1 = \varphi_2$.*

The following theorem is used to extract and insert an extra condition into both sides of an equivalence; note that it also holds for φ_1 on the right side, by the commutativity of conjunction.

Lemma 3. *Let Γ be a theory and φ_1, φ_2, and φ_3 be patterns. Then $\Gamma \vdash (\varphi_1 \land \varphi_2 \leftrightarrow \varphi_1 \land \varphi_3) \leftrightarrow (\varphi_1 \to \varphi_2 \leftrightarrow \varphi_3)$.*

Lemma 3 can be extended to work with equality using deduction:

Lemma 4. *For all theories Γ, predicate patterns φ_1, and patterns φ_2, φ_3, $\Gamma \vdash (\varphi_1 \land \varphi_2 = \varphi_1 \land \varphi_3) \leftrightarrow (\varphi_1 \to \varphi_2 = \varphi_3)$ holds.*

2.2 Unification

Unification [2] is the process of solving equations between symbolic expressions. The solution of the unification problem is a substitution that maps variables to expressions, which, when applied to the symbolic expressions, makes them syntactically equal (identical). In this subsection, we recall some essential concepts (mainly from [2]) which we will use when discussing unification in matching logic.

The following definitions assume that we work using terms (t) in a term algebra constructed with function symbols (f) and variables (x).

Definition 9 (Substitution in unification)**.** A substitution is a list of bindings $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$, mapping variables to terms. This is similar to the one in Definition 3, however, unlike that one, this version is not limited to a single variable and pattern.

The application of a substitution of a term ($t\sigma$) is defined as usual, in a recursive descent manner on applications.

Definition 10 (Composition of substitutions)**.** Let $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ and η be two substitutions. The composition of these, written as $\sigma\eta$ can be defined as $\sigma\eta \overset{\text{def}}{=} \{x_1 \mapsto t_1\eta, \dots, x_n \mapsto t_n\eta\}$, i.e. we apply the substitution η to every pattern on the right side of σ.

Next we define how to determine if two substitutions are equal.

Definition 11 (Equality of substitutions)**.** Two substitutions σ and η are equal if they are extensionally equal: $t\sigma = t\eta$, for all t, that is if their effect is the same when applied to any term.

We can combine these two operations to define when one substitution is more general than another.

Definition 12 (More general substitution)**.** We say that σ is more general than η, written as $\sigma \leq \eta$ if there is a substitution θ such that $\sigma\theta = \eta$.

The unification algorithm will produce a substitution that, when applied to the input terms, produces the same term. We call this substitution a unifier.

Definition 13 (Unifier, unifiable terms)**.** A substitution σ is called a unifier of two patterns t_1 and t_2 if $t_1\sigma = t_2\sigma$. If there exists a unifier of two terms, we call the terms unifiable.

However, there may be infinitely many unifiers that only differ in some insignificant details. For example, if the terms are $f\,x$ and $f\,y$, we could replace x and y with any term, but all that matters is that they need to be the same. To solve this, we introduce the concept of the most general unifier, using the more general relation from above.

Definition 14 (Most general unifier)**.** A unifier σ is called the most general if for any unifier θ, σ is more general than θ ($\sigma \leq \theta$).

It follows from the definition of the more general unifier that any other solution may be obtained from this by instantiation. In this paper we will not consider higher order unification (i.e. variables may not stand in for functions).

Definition 15 (Predicate of a substitution)**.** It is possible to extract a predicate from a substitution σ, denoted ϕ^σ. If $\sigma = \{x_1 \mapsto t_1,\, \dots,\, x_n \mapsto t_n\}$, then $\phi^\sigma \overset{\text{def}}{=} x_1 = t_1 \wedge \cdots \wedge x_n = t_n$.

In the following sections, we will solve semantic unification by solving syntactic unification, producing a most general unifier and turning it into a semantically unifying pattern in matching logic.

3 Related Work

3.1 Rule-based Unification

There are several algorithms for solving unification problems. We will focus on the rule-based approach that mimics the recursive descent algorithm computing the most general unifier and review the fundamental work presented in [2]. In particular, [2] introduces the so-called *unification problem*, which is either a set of pairs of term patterns, or a special symbol $\perp$, representing an unsolvable problem[4].

The algorithm is represented by a *unification step* relation, denoted as $P \Rightarrow P'$ for some P and P' unification problems. This relation is inductively defined by the rules of the algorithm. See Table 1 for the rules. The reflexive-transitive closure of this relation is written as $P \Rightarrow^* P'$. It is assumed that the left side of $\Rightarrow$, $\Rightarrow^*$, and $\cup$ is not $\perp$.

Delete	$P \cup \{(t,\, t)\} \Rightarrow P$
Decomposition	$P \cup \{(f(t_1,\, \dots,\, t_n),\, f(u_1,\, \dots,\, u_n))\} \Rightarrow P \cup \{(t_1,\, u_1),\, \dots,\, (t_n,\, u_n)\}$
Symbol clash	$P \cup \{(f(t_1,\, \dots,\, t_n),\, g(u_1,\, \dots,\, u_m))\} \Rightarrow \perp$
Orient	$P \cup \{(t,\, x)\} \Rightarrow P \cup \{(x,\, t)\}$ if $t \notin EV$
Occurs check	$P \cup \{(x,\, t)\} \Rightarrow \perp$ if $x \in FV(t)$
Elimination	$P \cup \{(x,\, t)\} \Rightarrow P\{x \mapsto t\} \cup \{(x,\, t)\}$ if $x \notin FV(t)$

Table 1: Rules of the unification algorithm

Definition 16 (Solved form)**.** A unification problem is considered to be in solved form if it is $\perp$ or a set $\{(x_1,\, t_1),\, \dots,\, (x_n,\, t_n)\}$, where $x_i \notin t_j$, for any $1 \leq i, j \leq n$ (i.e. the first term in the pairs is a variable that does not appear in the second term of any of the pairs).

[4]For brevity, from this point on we use $\perp$ do denote the failed unification problem instead the falsum pattern.

A unification problem that is in solved form and is not $\perp$ can be seen as a substitution. If we start with a unification problem with just the two terms we want to unify as a pair, then apply the rules of the algorithm, we will either end up with $\perp$, in which case the terms are not unifiable, or a unification problem whose corresponding substitution is the most general unifier of the terms. This expresses the *soundness* of the algorithm. Note that the opposite of the above statement is also true:

Theorem 3. If σ is the most general unifier of t_1 and t_2, then there exists a unification problem P such that $\{(t_1,t_2)\} \Rightarrow^* P$ and P is in solved form, with its corresponding substitution being σ.

Let us demonstrate rule-based unification with an example, which we will revisit in later sections for the sake of comparison.

Example 1. The following is an exhaustive application of the unification rules on the unification problem constructed with terms $f(x,\ g(1),\ g(z))$ and $f(g(y),\ g(y),\ g(g(x)))$.

$$\begin{aligned}
\{\underline{(f(x,\ g(1),\ g(z)),\ f(g(y),\ g(y),\ g(g(x))))}\} &\overset{\textbf{Decomposition}}{\Rightarrow} \\
\{(x,\ g(y)),\ \underline{(g(1),\ g(y))},\ (g(z),\ g(g(x)))\} &\overset{\textbf{Decomposition}}{\Rightarrow} \\
\{(x,\ g(y)),\ (1,\ y),\ \underline{(g(z),\ g(g(x)))}\} &\overset{\textbf{Decomposition}}{\Rightarrow} \\
\{(x,\ g(y)),\ \underline{(1,\ y)},\ (z,\ g(x))\} &\overset{\textbf{Orient}}{\Rightarrow} \\
\{(x,\ g(y)),\ \underline{(y,\ 1)},\ (z,\ g(x))\} &\overset{\textbf{Elimination}}{\Rightarrow} \\
\{\underline{(x,\ g(1))},\ (y,\ 1),\ (z,\ g(x))\} &\overset{\textbf{Elimination}}{\Rightarrow} \\
\{(x,\ g(1)),\ (y,\ 1),\ (z,\ g(g(1)))\} &
\end{aligned}$$

3.2 Unification in Matching Logic

In matching logic, symbolic expressions are encoded as term patterns that are made with applications of constructor function symbols. Unification has already been investigated in matching logic; in particular, Arusoaie et al. [1] adapted the results of [2] to the sorted, polyadic variant of matching logic (where applications are of form $f(x_1,\ \dots,\ x_n)$). In this paper, we do a similar adaptation, but to the unsorted, applicative variant of the logic, where application is a binary operation: $\varphi_1\ \varphi_2$.

It is to be noted that [1] argues about the soundness of unification on a semantic basis (in terms of the evaluations of patterns), while our work shows a syntactic proof of soundness, based on the sequent calculus for the logic, particularly exploiting the rules for deduction and rewriting.

Although the logic language and therefore the unification problems are somewhat different in our approach, we tried to reuse as much as possible from that of [1]. Specifically, even though we generalise the original unification problem into an abstract data type, we closely follow the approach of turning unification problems and substitutions into predicate patterns in matching logic.

4 Unification in an Applicative Matching Logic

As discussed in Section 3, the rule-based unification algorithm relies on a data structure called a unification problem. This is a set of pairs in the original presentation [2], but we generalise it to an abstract data type that is not tied to a specific underlying container.

4.1 Abstract Unification Problem

First of all, we outline the operations that can be abstracted into the constructors of the new type (and were inherited from set theory and substitution theory in the original definition). The constructors of abstract unification problems are as follows (P is a unification problem, t, t_1, t_2 are term patterns, x is a variable):

- $\langle t_1,\, t_2\rangle$ creates a new unification problem from a single pair of terms (this method is used to create the initial problem from the input terms that the algorithm is applied to);
- $\perp$ represents a unification problem that "failed" because the initial terms were not unifiable;
- $P \triangleleft \langle t_1,\, t_2\rangle$ is an insert operation that adds a single pair to the data structure;
- $P[t/x]$ transforms the data structure by substituting every occurrence of x in every pair by t.

We chose to define a constructor to create a singleton unification problem, but note that it would have also been equally correct to allow the creation of an empty problem $\emptyset$, in which case singleton could have simply been defined as $\langle t_1,\, t_2\rangle \stackrel{\text{def}}{=} \emptyset \triangleleft \langle t_1,\, t_2\rangle$. Although we rarely need to work with an empty unification problem, it may be created by the algorithm if the input terms are already the same, therefore no substitution is needed to unify them.

There is only a single destructor for the data structure, ϕ^P, which generates a pattern out of all the stored pairs. Intuitively, when the underlying representation uses sets and pairs, the implementation of decomposition would be defined by the following line:

$$\phi^P \stackrel{\text{def}}{=} \bigwedge_{(t_1,\, t_2)\in P} t_1 = t_2$$

However, abstract unification problems can be instantiated to various concrete representations. In the following, we provide the specification of the unification problem by determining the behaviour of the constructor and destructor operations, independent of the representation.

Injectivity. From now on, we suppose that Γ is a theory that contains the theory of definedness and includes axioms constraining all term constructor symbols to be modelled by injective functions; namely, we assume instances of the following axiom scheme for all constructor symbols f:

$$\forall x_1,\, \ldots,\, x_n,\, y_1,\, \ldots,\, y_n. f(x_1,\, \ldots,\, x_n) = f(y_1,\, \ldots,\, y_n) \to x_1 = y_1 \wedge \cdots \wedge x_n = y_n$$

Now, the following four specification axioms define the behaviour of the operations of abstract unification problems:

- The first property states that a singleton problem is equivalent to a single equality made from its constituent pair.

  Property 1. $\Gamma \vdash \phi^{\langle t_1,\, t_2\rangle} \leftrightarrow t_1 = t_2$
- The second property says that if a pair is inserted into the problem, its corresponding equality should appear in the predicate as well.

  Property 2. $\Gamma \vdash \phi^{P \triangleleft \langle t_1,\, t_2\rangle} \leftrightarrow t_1 = t_2 \wedge \phi^P$
- The third property states that substitution propagates through the predicate creation, that is substituting the terms in the data structure results in substituting them in the predicate as well.

  Property 3. $\Gamma \vdash \phi^{P[t/x]} \leftrightarrow (\phi^P)[t/x]$
- Finally, the fourth property expresses that if we extend a non-$\perp$ unification problem, the result will not be $\perp$.

  Property 4. $\Gamma \vdash P \neq \perp \to P \triangleleft \langle t_1,\, t_2\rangle \neq \perp$

4.2 Curried Rules for Abstract Unification Problems

While the related work [1] uses the polyadic version of the logic to define the rules of the unification algorithm, our goal is to create a formalism in the applicative version. Since the latter only has binary function application, every rule that uses this connective in the former must be rephrased. This is done by replacing functions with their curried versions, essentially taking the argument list and applying its elements one at a time:

$$f(t_1, \ldots, t_n) \longrightarrow (((f\, t_1) \ldots) t_n)$$

The rules impacted by this change are **Decomposition** and **Symbol clash**. The new ruleset is described in Table 2. Notice how, because we always insert one pair in most rules, and two pairs in **Decomposition**, the previously used union operations can be easily replaced with simple insertions.

Delete	$P \triangleleft \langle t,\, t\rangle \Rightarrow P$
Decomposition	$P \triangleleft \langle t_1 t_2,\, u_1 u_2\rangle \Rightarrow P \triangleleft \langle t_1,\, u_1\rangle \triangleleft \langle t_2,\, u_2\rangle$
Symbol clash L	$P \triangleleft \langle f,\, t\rangle \Rightarrow \bot \quad \text{if } t \neq f \wedge t \notin EV$
Symbol clash R	$P \triangleleft \langle t,\, f\rangle \Rightarrow \bot \quad \text{if } t \neq f \wedge t \notin EV$
Orient	$P \triangleleft \langle t,\, x\rangle \Rightarrow P \triangleleft \langle x,\, t\rangle \quad \text{if } t \notin EV$
Occurs check	$P \triangleleft \langle x,\, t\rangle \Rightarrow \bot \quad \text{if } x \in FV(t)$
Elimination	$P \triangleleft \langle x,\, t\rangle \Rightarrow P[t/x] \triangleleft \langle x,\, t\rangle \quad \text{if } x \notin FV(t)$

Table 2: The new rules for the unification algorithm.

In the case of **Decomposition**, it is now necessary to tackle functions one parameter at a time. We also need to add both sides to the unification problem, as it is no longer guaranteed that the left side is a function symbol, and it may contain further applications. Note that it is no longer possible to tell if the function symbols and their arity match until we fully decompose the application, and the symbols themselves will also be added to the set.

This is not a problem, however, because if the symbols and the arity did match, then the **Delete** rule can be used to get rid of the extra pair. If they did not match, we will end up with a symbol and either a different symbol or an application in the set. These are the two cases that the new **Symbol clash** rules solve. We have decided to split this rule because it is easier to state. They no longer deal with function applications, instead they are specialized to be used at the end of a chain of **Decomposition**s, filtering out cases that the original **Symbol clash** would have done in a single step. The $t \neq f$ condition plays a dual purpose here. In case the arities of the function symbols matched, but the symbols themselves did not, t will necessarily be some symbol g, that is different from f, satisfying the condition. If the arities did not match however, then t will be an application, which obviously will not be equal to a symbol, therefore satisfying the condition again. However, t may still be some variable x, and we must take care not to reject this symbol-variable pair using the new rules as those are not erroneous.

We demonstrate in Figure 4 with some small examples how the new rules compare to the old ones presented and explained in Section 3.

4.3 Soundness

In this section we present supporting lemmas similar to those discussed in [1]; however, we emphasise that they prove these results semantically, while we construct syntactical proofs. At the end of the section, we prove the soundness of the rule-based unification.

$$\begin{aligned} \{(f(x,\, a),\, f(b,\, y))\} &\overset{\textbf{Decomposition}}{\Rightarrow} \\ \{(x,\, b),\, (a,\, y)\} & \end{aligned}$$

$$\Big\downarrow$$

$$\begin{aligned} \langle f\,x\,a,\, f\,b\,y\rangle &\overset{\textbf{Decomposition}}{\Rightarrow} \\ \langle f\,x,\, f\,b\rangle \lhd \langle a,\, y\rangle &\overset{\textbf{Decomposition}}{\Rightarrow} \\ \langle f,\, f\rangle \lhd \langle x,\, b\rangle \lhd \langle a,\, y\rangle &\overset{\textbf{Delete}}{\Rightarrow} \\ \langle x,\, b\rangle \lhd \langle a,\, y\rangle & \end{aligned}$$

(a) Same arity and symbol

$$\begin{aligned} \{(f(x,\, a),\, g(b,\, y))\} &\overset{\textbf{Symbol clash}}{\Rightarrow} \\ \bot & \end{aligned}$$

$$\Big\downarrow$$

$$\begin{aligned} \langle f\,x\,a,\, g\,b\,y\rangle &\overset{\textbf{Decomposition}}{\Rightarrow} \\ \langle f\,x,\, g\,b\rangle \lhd \langle a,\, y\rangle &\overset{\textbf{Decomposition}}{\Rightarrow} \\ \langle f,\, g\rangle \lhd \langle x,\, b\rangle \lhd \langle a,\, y\rangle &\overset{\textbf{Symbol clash}}{\Rightarrow} \\ \bot & \end{aligned}$$

(b) Different symbol

$$\begin{aligned} \{(f(x,a),\, f(y))\} &\overset{\textbf{Symbol clash}}{\Rightarrow} \\ \bot & \end{aligned}$$

$$\Big\downarrow$$

$$\begin{aligned} \langle f\,x\,a,\, f\,y\rangle &\overset{\textbf{Decomposition}}{\Rightarrow} \\ \langle f\,x,\, f\rangle \lhd \langle a,\, y\rangle &\overset{\textbf{Symbol clash}}{\Rightarrow} \\ \bot & \end{aligned}$$

(c) Different arity

Figure 4: Comparison of old and new rules. Notice how $(a,\, y)$ and $(x,\, b)$ are not filtered out by either version of **Symbol clash** as they are not errors.

Lemma 5. *Let φ_1 and φ_2 be functional patterns. Then $\Gamma \vdash \varphi_1 \wedge \varphi_2 \leftrightarrow \varphi_1 \wedge (\varphi_1 = \varphi_2)$.*

Proof. First we prove the $\leftarrow$ direction. Here we simply split the premise and rewrite the goal using $(=\vdash)$ with $\varphi_1 = \varphi_2$ to obtain $\varphi_1 \wedge \varphi_1$, which is easily solved by the φ_1 hypothesis.

For the $\rightarrow$ direction, we know from Lemma 1that $\varphi \rightarrow \lceil \varphi \rceil$. We apply this to the $\varphi_1 \wedge \varphi_2$ hypothesis, that gives us a defined conjunction, which is the definition of $\in$, so our new hypothesis is $\varphi_1 \in \varphi_2$.

Then, we discharge φ_1 by using the left hand side of the original hypothesis, and $\varphi_1 \in \varphi_2$ implies $\varphi_1 = \varphi_2$ as shown in Lemma 2, so we can solve that part as well. $\square$

This lemma of utmost practical relevance, since it is used to set up the unification problem by introducing the equality pattern $\varphi_1 = \varphi_2$ which can be turned into the unification problem $\langle \varphi_1,\, \varphi_2 \rangle$ to solve.

Lemma 6. *Let φ and t be patterns, and x be a variable. Then $\Gamma \vdash x = t \rightarrow \varphi[t/x] = \varphi$.*

Proof. For technical reasons, we do not use the rewrite rule here, but the congruence of equality [7] on $x = t$, using the pattern context $\varphi[\square/x] = \varphi$. We can assume that $\square$ is a fresh variable in φ. This gives us $\varphi[\square/x][x/\square] = \varphi \leftrightarrow \varphi[\square/x][t/\square] = \varphi$.

In $\varphi[\square/x][x/\square]$ we replace all occurrences of x with a fresh variable $\square$, and then immediately replace all occurrences of that with x. It is clear to see that this operation is the same as $\varphi[x/x]$, which actually does nothing and so the pattern is equivalent to just φ.

We can make a similar argument in the case of $\varphi[\square/x][t/\square]$, however, this time it is t that we place back into $\square$, so this pattern is equivalent to $\varphi[t/x]$.

Rewriting the formula with these two equivalences, we get $\varphi = \varphi \leftrightarrow \varphi[t/x] = \varphi$.

As equivalence is just the conjunction of two implications, we can drop the $\leftarrow$ direction, keeping only the $\rightarrow$ one.

Finally, $\varphi = \varphi$ is proven by reflexivity, so by modus ponens on the implication, we arrive at our goal of $\varphi[t/x] = \varphi$. $\square$

The above lemma makes it easier to simplify substitutions using congruence in future proofs.

Lemma 7. *Let φ be a pattern and σ a substitution. Then $\Gamma \vdash \varphi\sigma \wedge \phi^{\sigma} \leftrightarrow \varphi \wedge \phi^{\sigma}$.*

Proof. First we extract the ϕ^{σ} from both sides of the equivalence using Lemma 3, which yields us $\phi^{\sigma} \rightarrow \varphi\sigma \leftrightarrow \varphi$.

Then we do induction on σ with any φ. If it is the empty substitution, we can prove $\top \rightarrow \varphi \leftrightarrow \varphi$ by reflexivity. If it is not, then there is at least one x element variable and t pattern that is part of σ ($\sigma = \{x \mapsto t\} \cup \sigma'$), and we also know that $x = t$ is part of ϕ^{σ} ($\phi^{\sigma} = (x = t) \wedge \phi^{\sigma'}$). Thus our goal becomes $x = t \wedge \phi^{\sigma'} \rightarrow \varphi[t/x]\sigma' \leftrightarrow \varphi$.

If we specialize the induction hypothesis with $\varphi[t/x]$ as φ, we get $\phi^{\sigma'} \rightarrow \varphi[t/x]\sigma' \leftrightarrow \varphi[t/x]$.

$\phi^{\sigma'}$ is part of our hypothesis. Using the transitivity and symmetry of equivalence with this and the goal, we can reduce the goal to $\varphi[t/x] \leftrightarrow \varphi$.

We still have $x = t$ in the hypothesis, and we can use Lemma 6 on it to obtain $\varphi[t/x] = \varphi$.

Finally, using DEDUCTION we can extract the equivalence from this equality, proving the goal. $\square$

This is, in a sense, an extension of Lemma 6 to set-based substitutions, with the added requirement for the predicate ϕ^{σ}.

Now we move on to justifying the soundness of single unification steps, and then argue about the correctness of unification step sequences by using induction.

Lemma 8. *Let P and P' be unification problems. If $P \Rightarrow P'$ and $P' \neq \bot$ then $\Gamma \vdash \phi^{P} \rightarrow \phi^{P'}$.*

Proof. We do induction on the $\Rightarrow$ relation. In most cases we will use Property 2, which states for every P unification problem and x, y terms that $\phi^{P \triangleleft \langle x,\, y\rangle} \leftrightarrow x = y \wedge \phi^{P}$.

- Using the property in the **Delete** case, we get $t = t \wedge \phi^{P} \rightarrow \phi^{P}$ which is trivially proven.
- In the case of **Decomposition**, we use the property three times, ultimately transforming the goal to $f\,t = g\,u \wedge \phi^{P} \rightarrow t = u \wedge f = g \wedge \phi^{P}$. This is the one time where we need to make use of the injectivity axiom, applying it to $f\,t = g\,u$ we obtain $f = g \wedge t = u$. With this now we can easily prove the goal.
- With **Symbol clash L**, **Symbol clash R**, and later **Occurs check** we actually have a contradiction, as in both cases $P' = \bot$ and we have a hypothesis that says $P' \neq \bot$, therefore these cases are immediately discharged.
- In the **Orient** case we use the property twice to transform the goal to $x = t \wedge \phi^{P} \rightarrow t = x \wedge \phi^{P}$ and this is solved using the symmetry of equality.

- Finally, in the case of **Elimination** we will once again use the property twice, however, this time that will not be enough. We can reduce the goal to $x = t \wedge \phi^P \to x = t \wedge \phi^{P[x/t]}$ and the $x = t$ part is solvable like before, but for the other one we need to take advantage of another property, Property 3. This says that $\phi^{P[x/t]} \leftrightarrow \phi^P[t/x]$. Rewriting with this, we get $x = t \wedge \phi^P \to x = t \wedge \phi^P[t/x]$. If we choose σ to be $\{x \mapsto t\}$ and φ to be ϕ^P, then this is a direct consequence of Lemma 7.

□

We can show that not only single steps but their sequence is correct:

Lemma 9. *If P and P' are unification problems, and $P \Rightarrow^* P'$, where $P' \neq \bot$, then* $\Gamma \vdash \phi^P \to \phi^{P'}$.

Proof. First we do induction on the $\Rightarrow^*$ relation.

In the reflexive case, we have $\phi^P \to \phi^P$, which is trivially provable.

In the transitive case, we know that there is a P'' such that $P \Rightarrow P''$ and $P'' \Rightarrow^* P'$. We also have an inductive hypothesis that if $P' \neq \bot$, then $\phi^{P''} \to \phi^{P'}$ and from the hypothesis we know that $P' \neq \bot$. We need to use these facts to prove that $\phi^P \to \phi^{P''}$.

For this we are going to use Lemma 8 with $P \Rightarrow P''$, but in order to do that, we first need to prove that $P'' \neq \bot$.

For that we are going to do another induction, this time on $P'' \Rightarrow^* P'$. In the reflexive case, when $P'' = P'$, we can once again use the $P' \neq \bot$ hypothesis to do this. In the transitive case we get that $P'' \Rightarrow P'''$, and we know that in every case of $\Rightarrow$, the left side is an insertion into some unification problem that is not $\bot$. That is, there is some P_0, such that $P_0 \neq \bot$ and $P'' = P_0 \lhd \langle \_, \_ \rangle$. But thanks to Property 4, that said $P_0 \neq \bot \to P_0 \lhd \langle \_, \_ \rangle \neq \bot$, we know that P'' cannot be $\bot$.

With this condition satisfied, we can use Lemma 8 to finish the proof. □

Corollary 1. If a substitution σ is the most general unifier of two patterns t_1 and t_2, then $\Gamma \vdash t_1 = t_2 \to \phi^\sigma$.

Proof. Here we can use Theorem 3 with the most general unifier σ to obtain a P' unification problem that is not $\bot$ and $\phi^{P'} = \phi^\sigma$. We also know that $t_1 = t_2$ is the same as $\phi^{\langle t_1, t_2 \rangle}$, from Property 1. By rewriting with these equalities, we have transformed the goal to be solvable by Lemma 9. □

Lemma 9 extends Lemma 8 to the reflexive-transitive closure of the relation, while Corollary 1 specializes it to the singleton pattern that the algorithm starts with.

Lemma 10. *Let σ be a unifier of two terms t_1 and t_2. Then* $\Gamma \vdash \phi^\sigma \to t_1 = t_2$.

Proof. First we take the ϕ^σ condition and add it to both sides of the equality using Lemma 4, resulting in $t_1 \wedge \phi^\sigma = t_2 \wedge \phi^\sigma$.

Next, we rewrite with Lemma 7 on both sides of the equality, which gives us $t_1\sigma \wedge \phi^\sigma = t_2\sigma \wedge \phi^\sigma$.

Now, we can remove the ϕ^σ from both sides by using Lemma 4 again, and the resulting $t_1\sigma = t_2\sigma$ is exactly the definition of σ being a unifier of t_1 and t_2, which is in our hypothesis. □

This is the same as the above, but in the other direction.

Theorem 4 (Soundness)**.** If a substitution σ is the most general unifier of two patterns t_1 and t_2, then $\Gamma \vdash t_1 = t_2 \leftrightarrow \phi^\sigma$.

Proof. This is the conjunction of Corollary 1 and Lemma 10. □

The soundness theorem derives a result that allows us to manipulate conjunction patterns so that the conjunction of two structural patterns can be turned into the conjunction of a single structural pattern and a single predicate pattern. This is of practical importance as it allows matching logic provers to extract predicates to be discharged by external solvers:

Corollary 2. Let σ be the most general unifier of two patterns t_1 and t_2. Then $\Gamma \vdash t_1 \wedge t_2 \leftrightarrow t_1 \wedge \phi^\sigma$ and $\Gamma \vdash t_1 \wedge t_2 \leftrightarrow t_2 \wedge \phi^\sigma$.

Proof. From Lemma 5 we know that $t_1 \wedge t_2 \leftrightarrow t_1 \wedge t_1 = t_2$. Using the commutativity of $\wedge$ and symmetry of $=$, we can also get $t_1 \wedge t_2 \leftrightarrow t_2 \wedge t_1 = t_2$ from the same lemma.

We also know from Theorem 4 that $t_1 = t_2 \leftrightarrow \phi^\sigma$. Rewriting the previous two statements with this we get the two statements of this theorem. □

Finally, we demonstrate the algorithm and its soundness on an example.

Example 2. We present the example shown in Example 1, generate its unifier and prove the first statement of Corollary 2 with its terms (the second may be proven similarly): There exists a σ substitution such that $\Gamma \vdash \underbrace{f\,x\,(g\,1)\,(g\,z)}_{t_1} \wedge \underbrace{f\,(g\,y)\,(g\,y)\,(g\,(g\,x))}_{t_2} \leftrightarrow \underbrace{f\,x\,(g\,1)\,(g\,z)}_{t_1} \wedge \phi^\sigma$.

We use existential quantification on σ because we know that the terms are unifiable (therefore it does exist) and it can be automatically inferred as we run the algorithm.

Proof. We begin with the $\rightarrow$ direction. First we use Lemma 5 to rewrite $t_1 \wedge t_2$ as $t_1 \wedge t_1 = t_2$.

Then, we use Lemma 9, since we do not have a proof that σ is the most general unifier (since we do not even know what σ is yet), we cannot use Corollary 1. For this, we need to prove that there is some non-$\bot$ P unification problem, such that $\langle t_1,\ t_2\rangle \Rightarrow^* P$. We will also not state what P is ahead of time, as that too can be inferred.

This proof follows the steps outlined in Example 1, but this time we need to use the applicative version of the **Decomposition** rule. We use the transitive constructor of $\Rightarrow^*$ with each step, and when we reach the solved form, end with the reflexive constructor. The steps are as follows:

$$\begin{array}{rc}
\langle \underline{f\,x\,(g\,1)\,(g\,z)},\ \underline{f\,(g\,y)\,(g\,y)\,(g\,(g\,x))}\rangle & \overset{\textbf{Decomposition}}{\Rightarrow} \\
\langle \underline{f\,x\,(g\,1)},\ \underline{f\,(g\,y)\,(g\,y)}\rangle \lhd \langle g\,z,\ g\,(g\,x)\rangle & \overset{\textbf{Decomposition}}{\Rightarrow} \\
\langle \underline{f\,x},\ \underline{f\,(g\,y)}\rangle \lhd \langle g\,1,\ g\,y\rangle \lhd \langle g\,z,\ g\,(g\,x)\rangle & \overset{\textbf{Decomposition}}{\Rightarrow} \\
\langle \underline{f},\ \underline{f}\rangle \lhd \langle x,\ g\,y\rangle \lhd \langle g\,1,\ g\,y\rangle \lhd \langle g\,z,\ g\,(g\,x)\rangle & \overset{\textbf{Delete}}{\Rightarrow} \\
\langle x,\ g\,y\rangle \lhd \langle \underline{g\,1},\ \underline{g\,y}\rangle \lhd \langle g\,z,\ g\,(g\,x)\rangle & \overset{\textbf{Decomposition}}{\Rightarrow} \\
\langle x,\ g\,y\rangle \lhd \langle \underline{g},\ \underline{g}\rangle \lhd \langle 1,\ y\rangle \lhd \langle g\,z,\ g\,(g\,x)\rangle & \overset{\textbf{Delete}}{\Rightarrow} \\
\langle x,\ g\,y\rangle \lhd \langle 1,\ y\rangle \lhd \langle \underline{g\,z},\ \underline{g\,(g\,x)}\rangle & \overset{\textbf{Decomposition}}{\Rightarrow} \\
\langle x,\ g\,y\rangle \lhd \langle 1,\ y\rangle \lhd \langle \underline{g},\ \underline{g}\rangle \lhd \langle z,\ g\,x\rangle & \overset{\textbf{Delete}}{\Rightarrow} \\
\langle x,\ g\,y\rangle \lhd \langle \underline{1},\ \underline{y}\rangle \lhd \langle z,\ g\,x\rangle & \overset{\textbf{Orient}}{\Rightarrow} \\
\langle x,\ g\,y\rangle \lhd \langle y,\ 1\rangle \lhd \langle z,\ g\,x\rangle &
\end{array}$$

We could have used **Elimination** at the end, however, it will not help us in this proof, so we omitted those steps of the original example. Now that we have successfully inferred what P is, we can easily prove that it is not $\bot$ and we have a hypothesis that says $t_1 = t_2 \rightarrow x = g\,y \wedge y = 1 \wedge z = g\,x$. We can apply this to our hypothesis of $t_1 = t_2$. With this, we can actually already solve our goal. The t_1 part is trivial, and the ϕ^σ part is unknown, because σ is uninstantiated, however, because it is a predicate, we know that it is a chain of conjunctions. If we solve this with the one from above, it will be inferred that σ is indeed $\{(x \mapsto gy),\ (y \mapsto 1),\ (z \mapsto gx)\}$. Now that we know this, the $\leftarrow$ direction can be solved. Here, the now known ϕ^σ gives us a series of equalities which we can use to repeatedly rewrite subterms in our goal of $t_1 \wedge t_2$ and the t_1 hypothesis, until the t_1 and t_2 in them become the same pattern. From this the solution is trivial.

We intuitively know that this rewriting is possible, as σ is a unifier of t_1 and t_2, therefore $t_1\sigma = t_2\sigma$, and although we did not formally prove this, it is easy to manually check in this example. In this manual proof it is also not necessary to do the substitution fully (i.e. to calculate the actual $t_1\sigma$ and $t_2\sigma$), it is enough to go until the two terms match. $\square$

4.4 Mechanization

As part of our work, we mechanized all the above presented results in the Coq proof assistant, based on an existing formalisation [3] of applicative matching logic. The formalisation uses a locally-nameless variable representation [4] and deep embedding, which makes the mathematical proofs and the formal proofs somewhat diverging here and there due to the additional technical complexity in the implementation stemming from well-formedness constraints. However, the formalisation provides an implementation of the sequent calculus in terms of an embedded proof mode [11], so our proof descriptions resemble the actual formal proof scripts [8].

Instances of the abstract unification problem. The abstract unification problem was implemented as a type class, and an instance of this class has been created for sets containing pairs of well-formed patterns, wrapped in an `option`. The optional type allows us to use the `None` value to represent failed unification problems. We are using stdpp's set classes [10], therefore this instance is still generic in the exact implementation of the set, as long as it has all required methods and properties. In the future, we plan to instantiate this class for lists and other containers and investigate their non-functional properties.

5 Conclusion

This paper presented a generalisation of the well-known unification problem of symbolic expressions. We defined abstract unification problems for term patterns in the applicative variant of matching logic, and following in the footsteps of the related work we defined a rule-based algorithm for solving abstract unification problems. We instantiated abstract problems to sets and demonstrated their behaviour via examples. Last but not least, we proved the soundness of the unification algorithm, syntactically, using a sequent calculus for matching logic. The entire development is supported by a mechanisation of the formal theory, implemented in the Coq proof assistant.

Acknowledgements

We warmly thank Runtime Verification Inc. for their generous funding support.

References

[1] Andrei Arusoaie & Dorel Lucanu (2019): *Unification in matching logic*. In Maurice H. ter Beek, Annabelle McIver & José N. Oliveira, editors: *Formal Methods – The Next 30 Years*, Springer International Publishing, Cham, pp. 502–518, doi:10.1007/978-3-030-30942-8_30.

[2] Franz Baader, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauss & Klaus Schulz (2001): *Chapter 8 - Unification theory*. In Alan Robinson & Andrei Voronkov, editors: *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, North-Holland, Amsterdam, pp. 445–533, doi:10.1016/B978-044450813-3/50010-2.

[3] Péter Bereczky, Xiaohong Chen, Dániel Horpácsi, Lucas Peña & Jan Tušil (2022): *Mechanizing matching logic in Coq*. *Electronic Proceedings in Theoretical Computer Science*, doi:10.4204/eptcs.369.2.

[4] Arthur Charguéraud (2012): *The Locally Nameless Representation*. *Journal of Automated Reasoning* 49(3), pp. 363–408, doi:10.1007/s10817-011-9225-2.

[5] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh & Grigore Roşu (2021): *Towards a trustworthy semantics-based language framework via proof generation*. In Alexandra Silva & K. Rustan M. Leino, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 477–499, doi:10.1007/978-3-030-81688-9_23.

[6] Xiaohong Chen, Dorel Lucanu & Grigore Roşu (2021): *Matching logic explained*. *Journal of Logical and Algebraic Methods in Programming* 120, p. 100638, doi:10.1016/j.jlamp.2021.100638.

[7] Xiaohong Chen & Grigore Rosu (2019): *Matching μ-Logic*. In: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1–13, doi:10.1109/LICS.2019.8785675.

[8] High Assurance Refactoring Project (2024): *Unification in applicative matching logic*. Available at `https://github.com/harp-project/AML-Formalization/releases/tag/v1.0.16`. Accessed on August 15th, 2024.

[9] Grigore Rosu (2017): *Matching logic*. *Logical Methods in Computer Science* Volume 13, Issue 4, doi:10.23638/LMCS-13(4:28)2017.

[10] Iris stdpp (2024): *Operations on sets*. Available at `https://plv.mpi-sws.org/coqdoc/stdpp/stdpp.base.html#lab26`. Accessed on August 2nd, 2024.

[11] Jan Tušil, Péter Bereczky & Dániel Horpácsi (2023): *Interactive Matching Logic Proofs in Coq*. In Erika Ábrahám, Clemens Dubslaff & Silvia Lizeth Tapia Tarifa, editors: *Theoretical Aspects of Computing – ICTAC 2023*, Springer Nature Switzerland, Cham, pp. 139–157, doi:10.1007/978-3-031-47963-2_10.

Abstract Continuation Semantics for Multiparty Interactions in Process Calculi based on CCS

Eneia Nicolae Todoran
Department of Computer Science
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
eneia.todoran@cs.utcluj.ro

Gabriel Ciobanu
Academia Europaea
London, United Kindom
https://www.ae-info.org
gabriel@info.uaic.ro

We develop denotational and operational semantics designed with continuations for process calculi based on Milner's CCS extended with mechanisms offering support for multiparty interactions. We investigate the abstractness of this continuation semantics. We show that our continuation-based denotational models are weakly abstract with respect to the corresponding operational models.

1 Introduction

In denotational semantics, continuations have a long tradition, being used to model a large variety of control mechanisms [20, 11, 12]. However, it is usually considered that continuations do not perform well enough as a tool for describing concurrent behaviour [16]. In [21, 6], we introduced a technique for denotational and operational semantic design named *continuation semantics for concurrency (CSC)* which can be used to handle advanced concurrent control mechanisms [9, 10, 22]. The distinctive characteristic of the CSC technique is the modelling of continuations as structured configurations of computations.

In this paper, we employ the CSC technique in providing denotational and operational semantics for the multiparty interaction mechanisms incorporated in two process calculi, namely CCS^n and CCS^{n+} [13], both based on the well-known CCS [15] – CCS^n and CCS^{n+} extend CCS with constructs called *joint input* and *joint prefix*, respectively, that can be used to express multiparty synchronous interactions. The semantic models are developed using the methodology of metric semantics [2].

In particular, we investigate the abstractness of continuation semantics. As it is known, the completeness condition of the full abstraction criterion [14] is often difficult to be fulfilled. In models designed with continuations, the problem may be even more difficult [4, 8]. Therefore, in [8, 22] we introduced a *weak abstractness* optimality criterion which preserves the correctness condition, but relaxes the completeness condition of the classic full abstractness criterion. The weak abstractness criterion comprises a weaker completeness condition called *weak completeness*, which is easier to establish because it needs to be checked only for denotable continuations (that handle only computations denotable by the language constructs and represent an invariant of the computation). We study the abstractness of continuation semantics based on the weak abstractness criterion. The continuation-based denotational models presented in this article for the multiparty interaction mechanisms incorporated in CCS^n and CCS^{n+} are weakly abstract with respect to the corresponding operational models.

Following the approach presented in [2], we start from the language $\mathscr{L}_{syn}$ given in Chapter 11 of [2]. As it is mentioned in [2], the language $\mathscr{L}_{syn}$ is "essentially based on CCS". Then, we consider two language named $\mathscr{L}_{CCS^n}$ and $\mathscr{L}_{CCS^{n+}}$ which extend the language $\mathscr{L}_{syn}$ with constructs for multiparty interactions: $\mathscr{L}_{CCS^n}$ extends $\mathscr{L}_{syn}$ with the joint input construct of CCS^n, and $\mathscr{L}_{CCS^{n+}}$ extends $\mathscr{L}_{syn}$ with the joint prefix construct of CCS^{n+}. We define and relate continuation-based denotational and operational semantics for $\mathscr{L}_{CCS^n}$ and $\mathscr{L}_{CCS^{n+}}$.

M. Marin, L. Leuştean (Eds.):
8th Symposium on Working Formal Methods (FROM 2024)
EPTCS 410, 2024, pp. 18–37, doi:10.4204/EPTCS.410.2

Contribution: By using the methodology of metric semantics, we develop original continuation semantics for the multiparty interaction mechanisms incorporated in CCS^n and CCS^{n+}. We provide a new representation of continuations based on a construction presented in Section 2.3. We show that the denotational models presented in this paper are weakly abstract with respect to the corresponding operational models. A weak abstractness result for CCS^n was also presented in [9]; the weak abstractness result for CCS^{n+} is new. The weak completeness condition of the weak abstractness principle presented in [8, 22] should be checked only for denotable continuations. Intuitively, the collection of denotable continuations have to be an invariant of the computation, in the sense that it is sufficiently large to support arbitrary computations denotable by program statements. The formal conditions capturing this intuition are studied initially in [8, 22]. In this article we offer a more general formal framework. We present the formal conditions which guarantee that the domain of denotable continuations is invariant under the operators used in the denotational semantics, where the domain of denotable continuations is the metric completion of the class of denotable continuations.

2 Preliminaries

We assume the reader is familiar with metric spaces, multisets, metric semantics [2], and the λ-calculus notation. For the used notions and notations, we refer the reader to [5, 6, 7, 8].

The notation $(x \in)X$ introduces the set X with typical element x ranging over X. We write $S \subseteq X$ to express that S is a subset of X. $|S|$ is the cardinal number of set S. Let X be a countable set.

The set of all finite multisets over X is represented by using the notation $[X]$; the construction $[X]$ and the operations on multisets that are specified formally in [7]. By a slight abuse, the cardinal number of a multiset $m \in [X]$ defined as $\sum_{x \in \mathsf{dom}\,(m)} m(x)$ is also denoted by $|m|$. Even though the same notation $|\cdot|$ is used regardless of whether '$\cdot$' is a set or a multiset, it is always evident from the context whether the argument '$\cdot$' is a set or a multiset. We represent a multiset by stringing its elements between square brackets '[' and ']'. For instance, the empty multiset is written as [], and $[e_1, e_2, e_2]$ is the multiset with one and two occurrences of the elements e_1 and e_2, respectively. If $f \in X \to Y$ is a function (with domain X and codomain Y) and S is a subset of X, $S \subseteq X$, the notation $f\restriction S$ denotes the function f restricted to the domain S, i.e. $f\restriction S : S \to Y$, $f\restriction S(x) = f(x), \forall x \in S$. Also, if $f \in X \to Y$ is a function, $\langle f \mid x \mapsto y \rangle : X \to Y$ is the function defined (for $x, x' \in X, y \in Y$) by: $\langle f \mid x \mapsto y \rangle(x') = \mathsf{if}\ x' = x\ \mathsf{then}\ y\ \mathsf{else}\ f(x')$. Given a function $f \in X \to X$, we say that an element $x \in X$ is a *fixed point* of f if $f(x) = x$, and if this fixed point is unique we write $x = \mathsf{fix}(f)$.

We present semantic models designed using the mathematical framework of *1-bounded complete metric spaces* [2]. We assume the following notions are known: *metric* and *ultrametric* space, *isometry* (between metric spaces, denoted by '$\cong$'), *Cauchy sequence*, *complete* metric space, *metric completion*, *compact* set, and the *discrete metric*. We use the notion of *metric domain* as a synonym for the notion of complete (ultra) metric space. We assume the reader is familiar with the standard metrics for defining composed metric structures [2]. We use the constructs for $\frac{1}{2}$-identity, disjoint union $(+)$, function space $(\to)$, Cartesian product $(\times)$, and the compact powerdomain. Every Cauchy sequence in a complete metric space M has a limit that is also in M. If (M_1, d_1) and (M_2, d_2) are metric spaces, a function $f : M_1 \to M_2$ is a *contraction* if $\exists c \in \mathbb{R}, 0 \le c < 1, \forall x, y \in M_1\ \ [d_2(f(x), f(x)) \le c \cdot d_1(x, y)]$. If $c = 1$ we say that f is *nonexpansive*; each nonexpansive function is *continuous* [2]. The set of all nonexpansive functions from M_1 to M_2 is denoted by $M_1 \xrightarrow{1} M_2$. We recall Banach's theorem.

Theorem 1 (Banach) *Let (M, d) be a non-empty complete metric space. Each contraction $f : M \to M$ has a* unique *fixed point.*

With $\mathscr{P}_{co}(\cdot)$ ($\mathscr{P}_{nco}(\cdot)$) we denote the power set of *compact* (*non-empty and compact*) subsets of '$\cdot$'. $\mathscr{P}_{fin}(\cdot)$ denotes the power set of *finite* subsets of '$\cdot$' (we always endow $\mathscr{P}_{fin}(\cdot)$ with the discrete metric).

Hereafter, we shall often suppress the metrics part in metric domain definitions. For example, we write $\frac{1}{2}\cdot M$ and $M_1 \times M_2$ instead of $(M, d_{\frac{1}{2}\cdot M})$ and $(M_1 \times M_2, d_{M_1\times M_2})$, respectively.

Let $(M,d), (M',d')$ be metric spaces. We write $(M,d) \lhd (M',d')$, or simply $M \lhd M'$, to express that M is a *subspace* of M', i.e, $M \subseteq M'$ and $d' \restriction M = d$ (the restriction of metric d' to M coincides with d).

For compact sets we use Theorem 2 (due to Kuratowski) and the characterization given in Theorem 3. Given a complete metric space (M,d) and a subset X, $X \subseteq M$, according to Theorem 3, the statement that X is compact is equivalent to the statement that X is the limit (with respect to Hausdorff metric d_H) of a sequence of finite sets [3]. The proofs of these theorems are also provided in [2].

Theorem 2 *[Kuratowski] Let (M,d) be a complete metric space.*

(a) If $(X_i)_i$ is a Cauchy sequence in $(\mathscr{P}_{nco}(M), d_H)$ then

$$\lim_i X_i = \{\lim_i x_i \mid \forall i : x_i \in X_i, (x_i)_i \text{ is a Cauchy sequence in } M\}.$$

(b) If $(X_i)_i$ is a Cauchy sequence in $(\mathscr{P}_{co}(M), d_H)$ then either, for almost all i, $X_i = \emptyset$, and $\lim_i X_i = \emptyset$, or for almost all i (say for $i \geq n$), $X_i \neq \emptyset$ and

$$\lim_i X_i = \{\lim_{i\geq n} x_i \mid \forall i \geq n : x_i \in X_i, (x_i)_i \text{ is a Cauchy sequence in } M\}.$$

(c) $(\mathscr{P}_{co}(M), d_H)$ and $(\mathscr{P}_{nco}(M), d_H)$ are complete metric spaces.

Theorem 3 *Let (M,d) be a complete metric space. A subset $X \subseteq M$ is compact whenever $X = \lim_i X_i$, where each X_i is a finite subset of M (the limit is taken with respect to the Hausdorff metric d_H).*

Remark 1 *(a) If M and M' are metric spaces with subspaces S and S' ($S \lhd M$ and $S' \lhd M'$), then $S + S' \lhd M + M'$, $S \times S' \lhd M \times M'$, $(A \to S) \lhd (A \to M)$, $\mathscr{P}_{co}(S) \lhd \mathscr{P}_{co}(M)$ and $\mathscr{P}_{nco}(S) \lhd \mathscr{P}_{nco}(M)$ (see [2], chapter 10).*

(b) Let $(M,d), (M_1,d_1)$ and (M_2,d_2) be metric spaces. It is easy to verify that, if $M_1 \lhd M$, $M_2 \lhd M$ and $M_1 \subseteq M_2$ then $M_1 \lhd M_2$.

Definition 1 *Given a metric space (M,d), a* completion *of (M,d) is a complete metric space $(\overline{M}, d')$ such that $M \lhd \overline{M}$ and for each element $x \in \overline{M}$ we have: $x = \lim_j x_j$, with $x_j \in M, \forall j \in \mathbb{N}$ (limit is taken with respect to metric d').*

Each metric space has a completion that is unique up to isometry [2]. For the proof of Remark 2, see [8].

Remark 2 *Let (M,d) be a complete metric space, and X be a subset of M, $X \subseteq M$. We use the notation $co(X|M)$ for the set $co(X|M) \overset{\text{not.}}{=} \{x \mid x \in M, x = \lim_i x_i, \forall i \in \mathbb{N} : x_i \in X, (x_i)_i$ is a Cauchy sequence in $X\}$, where limits are taken with respect to d (as (M,d) is complete $\lim_i x_i \in M$). If we endow X with $d_X = d \restriction X$ and $co(X|M)$ with $d_{co(X|M)} = d \restriction co(X|M)$, then $(co(X|M), d_{co(X|M)})$ is a metric completion of (X, d_X). It is easy to see that $X \lhd co(X|M)$ and $co(X|M) \lhd M$.*

Remark 3 *(a) If (M_1,d_1) and (M_2,d_2) are complete metric spaces and $(x_i)_i$ is a Cauchy sequence in $M_1 + M_2$, then for almost all i (i.e., for all but a finite number of exceptions) we have that $x_i = (1, x_i')$ or $x_i = (2, x_i')$, where $(x_i')_i$ is a Cauchy sequence in M_1 or M_2, respectively (see [2]).*

(b) Let (M_1,d_1) and (M_2,d_2) be complete metric spaces. If $(x_1^i, x_2^i)_i$ is a Cauchy sequence in $M_1 \times M_2$, then $(x_1^i)_i$ is a Cauchy sequence in M_1 and $(x_2^i)_i$ is a Cauchy sequence in M_2. Since M_1 and M_2 are complete, there exists $x_1 \in M_1$ and $x_2 \in M_2$ such that $x_1 = \lim_i x_1^i$ and $x_2 = \lim_i x_2^i$, and $\lim_i (x_1^i, x_2^i) = (x_1, x_2) = (\lim_i x_1^i, \lim_i x_2^i)$ [2].

(c) *Let (M,d) be a complete metric space. Let $(x_i)_i$ be a convergent sequence in M with limit $x = \lim_i x_i$. Then $(x_i)_i$ has a subsequence $(x_{f(i)})_i$ such that*

$$\forall n \, \forall j \geq n \, [d(x_{f(j)}, x) \leq 2^{-n}], \tag{1}$$

where $f : \mathbb{N} \to \mathbb{N}$ is a strictly monotone mapping, i.e., $f(i) < f(i')$ whenever $i < i'$. We obtain such a subsequence (by imposing the condition that the function $f : \mathbb{N} \to \mathbb{N}$ is strictly monotone and) by putting $f(0) = 0$, and if $i > 0$ then $f(i) = m$, where $m \in \mathbb{N}$ is the smallest natural number such that

$$\forall l \geq m \, [d(x_l, x) \leq 2^{-i}]. \tag{2}$$

It is easy to see that this subsequence satisfies property (1). *Clearly, if $n = 0$ (which implies $2^{-n} = 1$), then* (1) *holds. If $n > 0$ and $j \geq n$, then property* (1) *also holds because we infer (from* (2)*) that $d(x_{f(j)}, x) \leq 2^{-j} \leq 2^{-n}$.*

Lemma 1 *Let (M,d), (M_1,d_1) and (M_2,d_2) be complete metric spaces. Let S, S_1 and S_2 be subsets of M, M_1 and M_2, respectively, $S \subseteq M$, $S_1 \subseteq M_1$ and $S_2 \subseteq M_2$. Let A be an arbitrary set. Then*

(a) $co(S_1 + S_2 | M_1 + M_2) = co(S_1|M_1) + co(S_2|M_2)$,

(b) $co(S_1 \times S_2 | M_1 \times M_2) = co(S_1|M_1) \times co(S_2|M_2)$,

(c) $A \to co(S|M) = co(A \to S | A \to M)$,

(d) $\mathscr{P}_{co}(co(S|M)) = co(\mathscr{P}_{co}(S) | \mathscr{P}_{co}(M))$,

(e) $\mathscr{P}_{nco}(co(S|M)) = co(\mathscr{P}_{nco}(S) | \mathscr{P}_{nco}(M))$.

Proof. Clearly, $S_1 + S_2 \subseteq M_1 + M_2$, $S_1 \times S_2 \subseteq M_1 \times M_2$, $(A \to S) \subseteq (A \to M)$, $\mathscr{P}_{co}(S) \subseteq \mathscr{P}_{co}(M)$ and $\mathscr{P}_{nco}(S) \subseteq \mathscr{P}_{nco}(M)$ (see Remark 1(a)). The proof for part (a) follows by using Remark 3(a). The proof for part (e) is similar to the proof for part (d). We provide below the proofs for parts (b), (c) and (d).

(b) We have $co(S_1 \times S_2 | M_1 \times M_2) = \{(x_1,x_2) \mid (x_1,x_2) \in M_1 \times M_2, (x_1,x_2) = \lim_i (x_1^i, x_2^i),$
$(x_1^i, x_2^i)_i$ is a Cauchy sequence in $S_1 \times S_2\}$ [Remark 3(b)]
$= \{(x_1,x_2) \mid x_1 \in M_1, x_1 = \lim_i x_1^i, (x_1^i)_i$ is a Cauchy sequence in S_1,
$x_2 \in M_2, x_2 = \lim_i x_2^i, (x_2^i)_i$ is a Cauchy sequence in $S_2\}$
$= \{x_1 \mid x_1 \in M_1, x_1 = \lim_i x_1^i, (x_1^i)_i$ is a Cauchy sequence in $S_1\} \times$
$\{x_2 \mid x_2 \in M_2, x_2 = \lim_i x_2^i, (x_2^i)_i$ is a Cauchy sequence in $S_2\}$
$= co(S_1|M_1) \times co(S_2|M_2)$.

(c) Let $f \in A \to co(S|M)$ (the space $co(S|M)$ is complete, by Remark 2). We define a Cauchy sequence $(f_i)_i$ in $A \to S$ ($f_i \in A \to S$, for all $i \in \mathbb{N}$) as follows: for each $a \in A$, since $f(a) \in co(S|M)$, we consider a Cauchy sequence $(x_i^a)_i$ in S ($x_i^a \in S$ for all $i \in \mathbb{N}$) such that $\lim_i x_i^a = f(a)$. Without loss of generality, we may assume that $\forall i \geq n \, [d(x_i^a, f(a)) \leq 2^{-n}]$ for any $n \in \mathbb{N}$.

For all $i \in \mathbb{N}$, we define $f_i \in A \to S$ by $f_i(a) = x_i^a$, for each $a \in A$. One can check that $(f_i)_i$ is a Cauchy sequence in $A \to S$ and $\lim_i f_i = f$. By remarks 2 and 1(a), $A \to co(S|M) \lhd A \to M$, and so $f \in A \to M$. Therefore, $f \in co(A \to S | A \to M)$. Since f was arbitrarily selected, we obtain $A \to co(S|M) \subseteq co(A \to S | A \to M)$.

Next, let $f \in co(A \to S | A \to M)$. Then $f = \lim_i f_i$, where $(f_i)_i$ is a Cauchy sequence in $A \to S$. It is easy to verify that, since $(f_i)_i$ is a Cauchy sequence in $A \to S$, $(f_i(a))_i$ is a Cauchy sequence in S for each $a \in A$. Therefore, since $(f_i(a))_i$ is a Cauchy sequence in S, one can check that $\lim_i f_i(a) = f(a) \in M$ for each $a \in A$. Hence, $f \in A \to co(S|M)$, which means that we have $co(A \to S | A \to M) \subseteq A \to co(S|M)$. We conclude that $A \to co(S|M) = co(A \to S | A \to M)$.

(d) First, we observe that $\emptyset \in \mathscr{P}_{co}(co(S|M))$, and also $\emptyset \in co(\mathscr{P}_{co}(S)|\mathscr{P}_{co}(M))$.

Next, let $X \in co(\mathscr{P}_{co}(S)|\mathscr{P}_{co}(M))$, $X \neq \emptyset$. Since $X \in co(\mathscr{P}_{co}(S)|\mathscr{P}_{co}(M))$, then $X \in \mathscr{P}_{co}(M)$ and $X = \lim_i X_i$, where $(X_i)_i$ is a Cauchy sequence with $X_i \in \mathscr{P}_{co}(S)$ for all $i \in \mathbb{N}$. By Theorem 2 (assuming that $X_i \neq \emptyset$ for almost all i, say for $i \geq n$), we have

$$\begin{aligned} X = \lim_i X_i &= \{\lim_{i \geq n} x_i \mid \forall i \geq n : x_i \in X_i,\ (x_i)_{i=n}^{\infty} \text{ is a Cauchy sequence in } S\} \\ &= \{x \mid x \in M, x = \lim_{i \geq n} x_i, \forall i \geq n : x_i \in X_i,\ (x_i)_{i=n}^{\infty} \text{ is a Cauchy sequence in } S\} \\ &\subseteq \{x \mid x \in M, x = \lim_i x_i, \forall i \in \mathbb{N} : x_i \in S,\ (x_i)_i \text{ is a Cauchy sequence in } S\} = co(S|M). \end{aligned}$$

Since X is compact and $X \subseteq co(S|M)$, we have $X \in \mathscr{P}_{co}(co(S|M))$.
Therefore, $co(\mathscr{P}_{co}(S)|\mathscr{P}_{co}(M)) \subseteq \mathscr{P}_{co}(co(S|M))$.

The proof that $\mathscr{P}_{co}(co(S|M)) \subseteq co(\mathscr{P}_{co}(S)|\mathscr{P}_{co}(M))$ follows by using Theorem 3.

□

2.1 Denotable continuations

The completeness condition of the weak abstraction criterion presented in this paper uses a notion of *denotable continuation*. The class of denotable continuations represents an invariant of the computation, and its definition relies on a construction that employs a compliance notion in function spaces (presented in Definition 2). The class of denotable continuations is introduced formally in Definition 6.

Definition 2 *Let (M_1,d_1) and (M_2,d_2) be metric spaces. Let S_1 and S_2 be nonempty subsets of M_1 and M_2, respectively, $S_1 \subseteq M_1$, and $S_2 \subseteq M_2$. We define the metric space $(M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle, d_C)$ by:*

$$M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle = \{f \mid f \in M_1 \xrightarrow{1} M_2, (\forall x \in S_1 : f(x) \in S_2)\} \qquad d_C = d_F \restriction M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle,$$

where d_F is the standard metric defined on $M_1 \xrightarrow{1} M_2$ [2],[1] and d_C is the restriction of d_F to $M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle$. We say that $(M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle, d_C)$ is an $S_1 \to S_2$ compliant function space.

Clearly, $M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle$ is a subset of $M_1 \xrightarrow{1} M_2$: $M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle \subseteq M_1 \xrightarrow{1} M_2$ ($M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle$ contains all nonexpansive functions $f \in M_1 \xrightarrow{1} M_2$, that in addition satisfy the property: $(\forall x \in S_1 : f(x) \in S_2)$).

Remark 4 *As in Definition 2, let (M_1,d_1) and (M_2,d_2) be metric spaces. Let S_1 and S_2 be nonempty subsets of M_1 and M_2, respectively, $S_1 \subseteq M_1$, $S_2 \subseteq M_2$. One can establish the properties presented below.*

(a) $(M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle, d_C)$ is a subspace of $(M_1 \xrightarrow{1} M_2, d_F)$): $M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle \lhd M_1 \xrightarrow{1} M_2$.

(b) If (M_1,d_1) and (M_2,d_2) are ultrametric then $(M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle, d_C)$ is also an ultrametric space.

(c) The sets S_1 and S_2 can be endowed with the metrics $d_1 \restriction S_1$ and $d_2 \restriction S_2$, respectively. If the spaces (M_2,d_2) and $(S_2, d_2 \restriction S_2)$ are complete then $(M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle, d_C)$ is also a complete metric space.

Remark 5 *Let M_1 and M_2 be metric spaces, with subspaces S_1 and S_2 such that $S_1 \lhd M_1$ and $S_2 \lhd M_2$. Then we can construct the $S_1 \to S_2$ compliant space $(M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle, d_F \restriction M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle)$, and (since, by Remark 2, $co(S_i|M_i) \lhd M_i$, for $i = 1,2$) we can also construct the $co(S_1|M_1) \to co(S_2|M_2)$ compliant function space $(M_1\langle co(S_1|M_1)\rangle \xrightarrow{1} M_2\langle co(S_2|M_2)\rangle, d_F \restriction M_1\langle co(S_1|M_1)\rangle \xrightarrow{1} M_2\langle co(S_2|M_2)\rangle)$.*

Lemma 2 *Let (M_1,d_1) and (M_2,d_2) be complete metric spaces. Let S_1 and S_2 be nonempty subsets of M_1 and M_2 such that $S_1 \subseteq M_1$ and $S_2 \subseteq M_2$. If $f \in M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle$, then $f \in M_1\langle co(S_1|M_1)\rangle \xrightarrow{1} M_2\langle co(S_2|M_2)\rangle$.*

[1]The metric defined on $M_1 \xrightarrow{1} M_2$ is also presented in [5] (Definition 2.7).

Corollary 1 *Let (M_1, d_1) and (M_2, d_2) be complete metric spaces, and S_1 and S_2 be nonempty subsets of M_1 and M_2 ($S_1 \subseteq M_1$, $S_2 \subseteq M_2$). Then we have $M_1\langle S_1\rangle \xrightarrow{1} M_2\langle S_2\rangle \lhd M_1\langle co(S_1|M_1)\rangle \xrightarrow{1} M_2\langle co(S_2|M_2)\rangle$.*

Remark 6 *In the metric approach [2], a continuation-based denotational semantics $\mathscr{D} : L \to \mathbf{D}$ is a function which maps elements of a language L to values in a domain $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$, where $\mathbf{D}$ is the domain of* computations *(or* denotations*),* $\mathbf{C}$ *is the domain of* continuations *and* $\mathbf{R}$ *is a domain of final answers. Note that $\mathbf{D}$, $\mathbf{C}$ and $\mathbf{R}$ are metric domains, i.e., complete metric spaces. In general, the domain of continuations $\mathbf{C}$ is given by an equation of the form $\mathbf{C} = \cdots(\frac{1}{2} \cdot \mathbf{D})\cdots$, i.e., the definition of $\mathbf{C}$ depends on the domain $\mathbf{D}$. In this paper, we consider only denotational semantics designed using domain equations of the form $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ that have unique solutions (up to isometry $\cong$) [1], and we focus on ultrametric domains.* [2]

The *semantic operators* that are used in the definition of a denotational semantics $\mathscr{D} : L \to \mathbf{D}$ are *nonexpansive functions* that receive as arguments and yield as results values of various types, including (combinations of) the domains $\mathbf{D}$, $\mathbf{C}$ and $\mathbf{R}$.

Definition 3 *Let $\mathscr{D} : L \to \mathbf{D}$ be a continuation-based denotational semantics, where the semantic domain $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ is as in Remark 6. We define two classes of metric domains $\mathscr{A}t_{class}$ and $\mathscr{O}t_{class}$ for $\mathscr{D}$, class $\mathscr{A}t_{class}$ with typical element $\mathscr{A}t$ and class $\mathscr{O}t_{class}$ with typical element $\mathscr{O}t$:*

$$\mathscr{A}t ::= \mathbf{M} \mid \mathbf{D} \mid \mathbf{C} \mid \mathscr{A}t \times \mathscr{A}t \mid \mathscr{A}t + \mathscr{A}t \mid A \to \mathscr{A}t \mid \mathscr{P}_{co}(\mathscr{A}t) \mid \mathscr{P}_{nco}(\mathscr{A}t)\ ,$$
$$\mathscr{O}t ::= \mathscr{A}t \xrightarrow{1} \mathscr{A}t\ .$$

An element $\mathscr{A}t \in \mathscr{A}t_{class}$ is an argument type*, an element $\mathscr{O}t \in \mathscr{O}t_{class}$ is an* operator type.
Here $\mathbf{M}$ is an arbitrary metric domain (a complete metric space) that does not depend on either $\mathbf{D}$ or $\mathbf{C}$.[3] *A is an arbitrary set. The composed domains $\mathscr{A}t \in \mathscr{A}t_{class}$ and $\mathscr{O}t \in \mathscr{O}t_{class}$ are endowed with the standard metrics defined on the product space and the function space, respectively [2].*[4]

Remark 7 *Since in Definition 3,* $\mathbf{M}$*,* $\mathbf{D}$ *and* $\mathbf{C}$ *are complete spaces, any argument type $\mathscr{A}t \in \mathscr{A}t_{class}$ is a metric domain (a complete metric space). Also, any operator type $\mathscr{O}t \in \mathscr{O}t_{class}$ is a metric domain [2].*[5]

Note that the (restricted) function space $A \to \mathscr{A}t$ and the compact and non-empty and compact power-domain constructions $\mathscr{P}_{co}(\mathscr{A}t)$ and $\mathscr{P}_{nco}(\mathscr{A}t)$) [6] are not needed in the approach presented in this paper, and are rarely used in practice to specify argument types. The compact powerdomain constructions can be used to specify nondeterministic behaviour by using operators for nondeterministic scheduling [7]. In this paper, the specification of nondeterministic behaviour is given in the definition of the semantic operators for parallel composition and nondeterministic choice (without the need for nondeterministic schedulers). However, the class $\mathscr{A}t_{class}$ can be extended with other constructions, including the (restricted) function space $A \to \mathscr{A}t$ (where A is an arbitrary set), and the compact and non-empty and compact powerdomain constructions $\mathscr{P}_{co}(\mathscr{A}t)$ and $\mathscr{P}_{nco}(\mathscr{A}t)$).

[2]In the applications presented in this article, the domains $\mathbf{D}$, $\mathbf{C}$ and $\mathbf{R}$ are complete ultrametric spaces.

[3]In particular, $\mathbf{M}$ could be $\mathbf{R}$, $\mathbf{M} = \mathbf{R}$ (in case $\mathbf{R}$ does not depend on either $\mathbf{D}$ or $\mathbf{C}$). In the applications presented in this paper the final domain $\mathbf{R}$ does not depend on either $\mathbf{D}$ or $\mathbf{C}$. In general, domain $\mathbf{R}$ may depend on $\mathbf{D}$ (see chapter 18 of [2]), in which case $\mathbf{R}$ may need to be modelled as a more complex argument type.

[4]The metrics defined on composed spaces are also presented in [5] (Definition 2.7).

[5]The completeness properties of composed spaces are also presented in [5] (Remark 2.8).

[6]Since in practice continuations are finite structures, and since any finite set is compact [2], the compactness requirement is satisfied naturally in most applications.

[7]To give an example, for a nature inspired formalism [10], it is presented a denotational semantics that uses a nondeterministic scheduler mapping which yields a collection of schedules, where each schedule is a pair consisting of a denotation (computation) and a corresponding continuation.

Definition 4 *We consider a continuation-based denotational semantics $\mathscr{D} : L \to \mathbf{D}$, where the semantic domain is $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ as in Definition 3. Let $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{C}}$ be subspaces of domains $\mathbf{D}$ and $\mathbf{C}$ such that $\tilde{\mathbf{D}} \lhd \mathbf{D}$ and $\tilde{\mathbf{C}} \lhd \mathbf{C}$. For any argument type $\mathscr{A}t \in \mathscr{A}t_{class}$, we define the metric space $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ (using induction on the structure of $\mathscr{A}t$) by:*

$$\mathbf{M}\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = \mathbf{M} \qquad \mathbf{D}\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = \tilde{\mathbf{D}} \qquad \mathbf{C}\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = \tilde{\mathbf{C}}$$
$$(\mathscr{A}t_1 \times \mathscr{A}t_2)\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = (\mathscr{A}t_1\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle) \times (\mathscr{A}t_2\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle).$$
$$(\mathscr{A}t_1 + \mathscr{A}t_2)\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = (\mathscr{A}t_1\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle) + (\mathscr{A}t_2\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle)$$
$$(A \to \mathscr{A}t)\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = A \to (\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle)$$
$$(\mathscr{P}_{co}(\mathscr{A}t))\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = \mathscr{P}_{co}(\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle)$$
$$(\mathscr{P}_{nco}(\mathscr{A}t))\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = \mathscr{P}_{nco}(\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle).$$

We use a similar notation for operator types $\mathscr{O}t \in \mathscr{O}t_{class}$. Namely, if $\mathscr{O}t = \mathscr{A}t_1 \xrightarrow{1} \mathscr{A}t_2$ (with $\mathscr{A}t_1, \mathscr{A}t_2 \in \mathscr{A}t_{class}$), we define the space $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ by:

$$\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = (\mathscr{A}t_1 \xrightarrow{1} \mathscr{A}t_2)\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle = \mathscr{A}t_1\langle\mathscr{A}t_1\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle\rangle \xrightarrow{1} \mathscr{A}t_2\langle\mathscr{A}t_2\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle\rangle.$$

For any $\mathscr{A}t \in \mathscr{A}t_{class}$ and $\mathscr{O}t \in \mathscr{O}t_{class}$ we have $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle \lhd \mathscr{A}t$ and $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle \lhd \mathscr{O}t$ (Remark 8), and we endow the spaces $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ and $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ with the metrics $d_{\mathscr{A}t}\restriction\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ and $d_{\mathscr{O}t}\restriction\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$, respectively.

Remark 8 *Let $\mathscr{D} : L \to \mathbf{D}$ with $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ be a continuation-based denotational semantics, as in Definition 3. Let $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{C}}$ be subspaces of domains $\mathbf{D}$ and $\mathbf{C}$, respectively, $\tilde{\mathbf{D}} \lhd \mathbf{D}$ and $\tilde{\mathbf{C}} \lhd \mathbf{C}$. Let $\mathscr{A}t \in \mathscr{A}t_{class}$ and $\mathscr{O}t \in \mathscr{O}t_{class}$.*

- *(a) The spaces $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ and $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ are well-defined, $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle \lhd \mathscr{A}t$ and $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle \lhd \mathscr{O}t$.*
- *(b) Assuming that spaces $\mathbf{M}$, $\mathbf{D}$ and $\mathbf{C}$ are ultrametric in Definition 3, $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ and $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ are also ultrametric spaces.*
- *(c) Furthermore, if the spaces $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{C}}$ are complete, then $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ and $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$ are also complete metric spaces.*

Lemma 3 *Let $\mathscr{D} : L \to \mathbf{D}$ with $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ be a continuation-based denotational semantics (as in Definition 3). Let $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{C}}$ be subspaces of domains $\mathbf{D}$ and $\mathbf{C}$ such that $\tilde{\mathbf{D}} \lhd \mathbf{D}$ and $\tilde{\mathbf{C}} \lhd \mathbf{C}$.*
For all $\mathscr{A}t \in \mathscr{A}t_{class}$, we have $\mathscr{A}t\langle co(\tilde{\mathbf{D}}|\mathbf{D}), co(\tilde{\mathbf{C}}|\mathbf{C})\rangle = co(\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle|\mathscr{A}t)$.

Definition 5 *We consider a continuation-based denotational semantics $\mathscr{D} : L \to \mathbf{D}$, where the semantic domain is $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ as in Definition 3. Let $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{C}}$ be subspaces of domains $\mathbf{D}$ and $\mathbf{C}$ such that $\tilde{\mathbf{D}} \lhd \mathbf{D}$ and $\tilde{\mathbf{C}} \lhd \mathbf{C}$. Let $\mathscr{O}t$ be an operator type, $\mathscr{O}t \in \mathscr{O}t_{class}$. Let $f \in \mathscr{O}t$ be an operator of type $\mathscr{O}t$. We say that the class of continuations $\tilde{\mathbf{C}}$ is* invariant *for $\tilde{\mathbf{D}}$ under the operator f iff $f \in \mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle$.*

Lemma 4 *Let $\mathscr{D} : L \to \mathbf{D}$ with $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ be a continuation-based denotational semantics (as in Definition 3). Let $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{C}}$ be subspaces of domains $\mathbf{D}$ and $\mathbf{C}$ such that $\tilde{\mathbf{D}} \lhd \mathbf{D}$ and $\tilde{\mathbf{C}} \lhd \mathbf{C}$.*

- *(a) $\mathscr{A}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle \lhd \mathscr{A}t\langle co(\tilde{\mathbf{D}}|\mathbf{D}), co(\tilde{\mathbf{C}}|\mathbf{C})\rangle$ for any $\mathscr{A}t \in \mathscr{A}t_{class}$.*
- *(b) $\mathscr{O}t\langle\tilde{\mathbf{D}},\tilde{\mathbf{C}}\rangle \lhd \mathscr{O}t\langle co(\tilde{\mathbf{D}}|\mathbf{D}), co(\tilde{\mathbf{C}}|\mathbf{C})\rangle$ for any $\mathscr{O}t \in \mathscr{O}t_{class}$.*
- *(c) Let $\mathscr{O}t \in \mathscr{O}t_{class}$ be an operator type, and let $f \in \mathscr{O}t$ be an operator of type $\mathscr{O}t$. If $\tilde{\mathbf{C}}$ is invariant for $\tilde{\mathbf{D}}$ under the operator $f \in \mathscr{O}t$, then $co(\tilde{\mathbf{C}}|\mathbf{C})$ is also invariant for $co(\tilde{\mathbf{D}}|\mathbf{D})$ under operator f.*

Definition 6 *Let $\mathscr{D} : L \to \mathbf{D}$ with $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ be a continuation-based metric denotational semantics. We put $\mathbf{D}^{\mathscr{D}} = \{\mathscr{D}(s) \mid s \in L\}$. Since $\mathscr{D}(s) \in \mathbf{D}$ for any $s \in L$, $\mathbf{D}^{\mathscr{D}} \lhd \mathbf{D}$ (we endow $\mathbf{D}^{\mathscr{D}}$ with $d_{\mathbf{D}}\restriction\mathbf{D}^{\mathscr{D}}$). Let $f_1 \in \mathscr{O}t_1, \cdots, f_n \in \mathscr{O}t_n$ be all operators used in the definition of the denotational mapping $\mathscr{D}$. If $\mathbf{C}^{\mathscr{D}}$ is* *a subspace of* $\mathbf{C}$*, namely* $\mathbf{C}^{\mathscr{D}} \lhd \mathbf{C}$*, we say that* $\mathbf{C}^{\mathscr{D}}$ *is a* class of denotable continuations *for* $\mathscr{D}$ *iff* $\mathbf{C}^{\mathscr{D}}$ *is invariant for* $\mathbf{D}^{\mathscr{D}}$ *under all operators* f_i $(i = 1,\ldots,n)$ *used in defining* $\mathscr{D}$*. If* $\mathbf{C}^{\mathscr{D}}$ *is a class of denotable continuations for* $\mathscr{D}$*, the metric domain* $co(\mathbf{C}^{\mathscr{D}}|\mathbf{C})$ *is called a* domain of denotable continuations *for* $\mathscr{D}$*.*

Remark 9 *Let* $\mathscr{D} : L \to \mathbf{D}$ *with* $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ *be a continuation-based metric denotational semantics, and* $\mathbf{D}^{\mathscr{D}} = \{\mathscr{D}(s) \mid s \in L\}$ *as in Definition 6. Let* $f_1 \in \mathscr{O}t_1, \cdots, f_n \in \mathscr{O}t_n$ *be all operators used in the definition of the denotational mapping* $\mathscr{D}$*. If* $\mathbf{C}^{\mathscr{D}}$ *is a class of denotable continuations for* $\mathscr{D}$ *and* $co(\mathbf{C}^{\mathscr{D}}|\mathbf{C})$ *is the corresponding metric domain of denotable continuations, by Lemma 4(c),* $co(\mathbf{C}^{\mathscr{D}}|\mathbf{C})$ *is invariant for* $co(\mathbf{D}^{\mathscr{D}}|\mathbf{D})$ *under all operators* f_i $(i = 1,\ldots,n)$ *used in the definition of* $\mathscr{D}$*.*

2.2 Weak abstractness criterion

If we compare the classic *full abstractness* criterion [14] with the *weak abstractness* criterion [8] employed in this paper, we emphasize that the *correctness* condition of the two criteria coincides, but the weak abstractness criterion relies on a weaker completeness condition called *weak completeness*, a condition that should be verified *only for denotable continuations*. While the classic full abstractness condition cannot be established in continuation semantics [4, 8], the abstractness of a continuation-based denotational model can be investigated based on the weak abstractness criterion. The terminology used here to present the abstraction criteria (comprising also the notion of a *syntactic context*) is taken from [2]. We recall the *completeness* condition of the full abstractness criterion for such a continuation-based model[8]. For this, we consider a language L, a continuation-based denotational semantics $\mathscr{D} : L \to \mathbf{D}$ (where the domain $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ is as in Remark 6 and $\mathbf{C}$ is the domain of continuations), and an operational semantics $\mathscr{O} : L \to \mathbf{O}$. If S ranges over a set of *syntactic contexts* for L, $\mathscr{D}$ is *complete* with respect to $\mathscr{O}$ when

$$\forall x_1,x_2 \in L\,[(\exists \gamma \in \mathbf{C}\,[\mathscr{D}(x_1)(\gamma) \neq \mathscr{D}(x_2)(\gamma)]) \Rightarrow (\exists S\,[\mathscr{O}(S(x_1)) \neq \mathscr{O}(S(x_2))])].$$

When the domain of continuations $\mathbf{C}$ contains elements which do not correspond to language elements, this completeness condition may not hold [4, 8]. Definition 7 presents the weak abstractness criterion which comprises a weaker completeness condition. In Definition 7 and Lemma 5, we assume that $(x \in)L$ is a language, $\mathscr{D} : L \to \mathbf{D}$ is a continuation-based denotational semantics where the denotational domain $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ is as in Remark 6, $(\gamma \in)\mathbf{C}$ is the domain of continuations, $\mathscr{O} : L \to \mathbf{O}$ is an operational semantics for L, and S is a typical element of the set of *syntactic contexts* for L.

Definition 7 *(Weak abstractness for continuation semantics)*

(a) $\mathscr{D}$ *is* correct *with respect to* $\mathscr{O}$ *iff* $\forall x_1,x_2 \in L[\mathscr{D}(x_1) = \mathscr{D}(x_2) \Rightarrow \forall S[\mathscr{O}(S(x_1)) = \mathscr{O}(S(x_2))]]$.

(b) If $\mathbf{C}^{\mathscr{D}}$ *is a class of denotable continuations for* $\mathscr{D}$ *and* $\overline{\mathbf{C}}^{\mathscr{D}} = co(\mathbf{C}^{\mathscr{D}}|\mathbf{C})$ *is the corresponding domain of denotable continuations for* $\mathscr{D}$*, then we say that* $\mathscr{D}$ *is* weakly complete w.r.t $\mathscr{O}$ and $\overline{\mathbf{C}}^{\mathscr{D}}$ *iff*

$$\forall x_1,x_2 \in L\,[(\exists \gamma \in \overline{\mathbf{C}}^{\mathscr{D}}\,[\mathscr{D}(x_1)\gamma \neq \mathscr{D}(x_2)\gamma]) \Rightarrow (\exists S\,[\mathscr{O}(S(x_1)) \neq \mathscr{O}(S(x_2))])].$$

We say that $\mathscr{D}$ *is* weakly complete with respect to $\mathscr{O}$ *iff there exists a class of denotable continuations* $\mathbf{C}^{\mathscr{D}}$ *such that* $\mathscr{D}$ *is* weakly complete *with respect to* $\mathscr{O}$ *and* $\overline{\mathbf{C}}^{\mathscr{D}}$*, where* $\overline{\mathbf{C}}^{\mathscr{D}} = co(\mathbf{C}^{\mathscr{D}}|\mathbf{C})$ *is the corresponding domain of denotable continuations.*

(c) $\mathscr{D}$ *is* weakly abstract *with respect to* $\mathscr{O}$ *iff* $\mathscr{D}$ *is correct and weakly complete with respect to* $\mathscr{O}$*.*

Lemma 5 *Let* $\mathscr{D} : L \to \mathbf{D}$ *be a continuation-based denotational semantics, where the domain* $\mathbf{D}$ *is given by* $\mathbf{D} \cong \mathbf{C} \xrightarrow{1} \mathbf{R}$ *and* $\mathbf{C}$ *is the domain of continuations (as in Definition 7). If* $\mathbf{C}^{\mathscr{D}}$ *is a class of denotable*

[8]The *correctness* condition of the two criteria coincides, and is presented in Definition 7(a).

continuations for $\mathscr{D}$ and $\overline{\mathbf{C}}^{\mathscr{D}} = co(\mathbf{C}^{\mathscr{D}}|\mathbf{C})$ *is the corresponding domain of denotable continuations for $\mathscr{D}$, then $\mathscr{D}$ is* weakly complete *with respect to $\mathscr{O}$ and* $\overline{\mathbf{C}}^{\mathscr{D}}$ *iff*

$$\forall x_1, x_2 \in L\,[(\exists \gamma \in \mathbf{C}^{\mathscr{D}}\,[\mathscr{D}(x_1)\gamma \neq \mathscr{D}(x_2)\gamma]) \Rightarrow (\exists S\,[\mathscr{O}(S(x_1)) \neq \mathscr{O}(S(x_2))])]. \tag{3}$$

Therefore, if there exists a class of denotable continuations $\mathbf{C}^{\mathscr{D}}$ *for $\mathscr{D}$ such that condition* (3) *is satisfied, then $\mathscr{D}$ is* weakly complete *with respect to $\mathscr{O}$.* The proof of Lemma 5 is provided in [8].

Since Remark 9 and Lemma 5 automatically extend the above properties to the entire domain of denotable continuations, it is enough to verify the invariance and completeness properties required by the weak abstraction criterion for the class of denotable continuations.

2.3 Finite bags and the structure of continuations

In this paper, the structure of continuations is defined based on a construction for finite bags $\langle\!| \cdot |\!\rangle$, which in turn is defined based on a set of identifiers Id. The symbols ; , $\|$ and $\backslash$ occurring in an identifier $\alpha \in Id$ are used to describe the semantics of sequential composition, parallel composition and restriction operators, respectively.

Definition 8 *(Identifiers) In the sequel of the paper we use a set $(c \in)\mathscr{N}$ of* names*; the set of names $\mathscr{N}$ is assumed to be countable, as in CCS [15]. We introduce a set of* identifiers $(\alpha \in)Id$ *given in BNF by:*

$$\alpha ::= \bullet \mid (;\bullet) \mid (\alpha\,;) \mid (\alpha\backslash c) \mid (\alpha \,\|) \mid (\|\, \alpha).$$

For substituting the hole symbol $\bullet$ occurring in an identifier α with α', we use the notation $\alpha(\alpha')$ given by: $\bullet(\alpha') = \alpha'$, $(;\bullet)(\alpha') = (;\bullet)$, $(\alpha\,;)(\alpha') = (\alpha(\alpha')\,;)$, $(\alpha\backslash c)(\alpha') = (\alpha(\alpha')\backslash c)$, $(\alpha\,\|)(\alpha') = (\alpha(\alpha')\,\|)$, *and* $(\|\,\alpha)(\alpha') = (\|\,\alpha(\alpha'))$.

The symbol $\bullet$ is used as a reference to an *active computation*. Thus, the substitution $\alpha(\alpha')$ does not replace the symbol $\bullet$ when it occurs on the right-hand side of a sequential composition $(;\bullet)$.

In previous works based on the CSC (continuation semantics for concurrency) technique, the set of identifiers is defined as a collection of finite sequences $\{1,2\}^*$ or $(\mathscr{N} \cup \{1,2\})^*$ endowed with a partial ordering relation that is used to express the structure of continuations [21, 6] and [9], respectively. In this paper, we employ a new representation of continuations based on the set of identifiers Id introduced in Definition 8.

We define and use the predicate $match_\alpha : (Id \times Id) \to Bool$ given by:

$match_\alpha(\bullet, \alpha) = \mathsf{true}$, $\quad match_\alpha((;\bullet),(;\bullet)) = \mathsf{true}$,

$match_\alpha((\alpha_1\,;),(\alpha_2\,;)) = match_\alpha((\alpha_1\,\|),(\alpha_2\,\|)) = match_\alpha((\|\,\alpha_1),(\|\,\alpha_2)) = match_\alpha(\alpha_1,\alpha_2)$,

$match_\alpha((\alpha_1\backslash c),(\alpha_2\backslash c)) = match_\alpha(\alpha_1,\alpha_2)$, and $match_\alpha(\alpha_1,\alpha_2) = \mathsf{false}$ otherwise.

By structural induction on α, one can show that $match_\alpha(\alpha,\alpha)$, $match_\alpha(\alpha,\alpha') \wedge match_\alpha(\alpha',\alpha'') \Rightarrow match_\alpha(\alpha,\alpha'')$, and $match_\alpha(\alpha,\alpha') \wedge match_\alpha(\alpha',\alpha) \Rightarrow \alpha = \alpha'$, for any $\alpha,\alpha',\alpha'' \in Id$. Thus, the relation $\leq = \{(\alpha,\alpha') \mid match_\alpha(\alpha,\alpha')\} (\subseteq Id \times Id)$ is a *partial order*. We write $\alpha \leq \alpha'$ to express that $(\alpha,\alpha') \in \leq$.

When we have $\alpha \leq \alpha'$, the identifier α' can be obtained from α by a substitution $\alpha' = \alpha(\alpha'')$ for some $\alpha'' \in Id$. Let $(\pi \in)\Pi = \mathscr{P}_{fin}(Id)$. For any $\pi \in \Pi$ and $\alpha \in Id$, we use the notation:

$$\pi|_\alpha = \{\alpha' \mid \alpha' \in \pi, \alpha \leq \alpha'\}.$$

We also define the operators $g : (Id \times Id) \to Id$ and $\ominus : (Id \times Id) \to (Id \cup \{\uparrow\})$ with $\uparrow \notin Id$ (for $\ominus$ we use the infix notation), as well as the predicate $in_\alpha : (\mathscr{N} \times Id) \to Bool$ by:

$g(\alpha,\alpha) = \alpha$, $\quad g((\alpha_1\,;),(\alpha_2\,;)) = (g(\alpha_1,\alpha_2)\,;)$, $\quad g((\alpha_1\backslash c),(\alpha_2\backslash c)) = (g(\alpha_1,\alpha_2)\backslash c)$,

$$g((\alpha_1 \parallel),(\alpha_2 \parallel)) = (g(\alpha_1,\alpha_2) \parallel),\quad g((\parallel \alpha_1),(\parallel \alpha_2)) = (\parallel g(\alpha_1,\alpha_2)),\ \text{and } g(\alpha_1,\alpha_2) = \bullet \ \text{ otherwise}$$

$$\alpha \ominus \bullet = \alpha,\quad (\alpha_1\,;) \ominus (\alpha_2\,;) = (\alpha_1 \parallel) \ominus (\alpha_2 \parallel) = (\parallel \alpha_1) \ominus (\parallel \alpha_2) = \alpha_1 \ominus \alpha_2,$$

$$(\alpha_1 \backslash c) \ominus (\alpha_2 \backslash c) = \alpha_1 \ominus \alpha_2, \quad \text{and} \quad \alpha_1 \ominus \alpha_2 = \uparrow \quad \text{otherwise}$$

$$in_\alpha(c,\bullet) = \mathsf{false},\quad in_\alpha(c,(;\bullet)) = \mathsf{false},\quad in_\alpha(c,(\alpha\backslash c')) = \ \mathsf{if}\ c' = c\ \mathsf{then\ true\ else}\ in_\alpha(c,\alpha),$$

$$in_\alpha(c,(\alpha\,;)) = in_\alpha(c,(\alpha \parallel)) = in_\alpha(c,(\parallel \alpha)) = in_\alpha(c,\alpha).$$

One can show that $g(\alpha_1,\alpha_2)$ is the *greatest lower bound* of α_1 and α_2 with respect to $\leq$.[9] For example, considering the identifiers $\alpha_1 = ((\alpha_1' \parallel)\backslash c)$ and $\alpha_2 = ((\parallel \alpha_2')\backslash c)$, we have $g(\alpha_1,\alpha_2) = (\bullet\backslash c)$. Clearly, $g(\alpha_1,\alpha_2) \leq \alpha_1$ and $g(\alpha_1,\alpha_2) \leq \alpha_2$. Moreover, one can show that, if $\alpha \leq \alpha'$ then $\alpha' \ominus \alpha \in Id$. For example, $((\alpha_1' \parallel)\backslash c) \ominus (\bullet\backslash c) = (\alpha_1' \parallel)$. $in_\alpha(c,\alpha) = \mathsf{true}$ if c occurs restricted in α.

We also define the predicate $\iota_2 : ((\mathscr{N} \times Id) \times (\mathscr{N} \times Id)) \to Bool$ by:

$$\begin{aligned}\iota_2(c_1@\alpha_1, c_2@\alpha_2) = \ & \mathsf{let}\ \alpha = g(\alpha_1,\alpha_2), \alpha_1' = \ominus(\alpha_1,\alpha), \alpha_2' = \ominus(\alpha_2,\alpha)\\ & \mathsf{in}\ (\alpha_1 \neq \alpha_2) \wedge (c_1 = c_2) \wedge \neg(in_\alpha(c_1,\alpha_1')) \wedge \neg(in_\alpha(c_2,\alpha_2')).\end{aligned}$$

We write a pair $(c,\alpha) \in \mathscr{N} \times Id$ as $c@\alpha$ to express that action c is executed by a process with identifier α. We model multiparty interactions using the binary interaction predicate $\iota_2(c_1@\alpha_1, c_2@\alpha_2)$; such an interaction is successful if the two actions are executed by different processes $\alpha_1 \neq \alpha_2$, they use the same interaction channel $c_1 = c_2$, and they are not disabled by restriction operators $(\cdot\backslash c)$.

As in [8, 9], we use the construct $\langle\!| \cdot |\!\rangle$ given below to model finite bags (multisets) of computations. Let $(x \in)\mathbf{X}$ be a metric domain, i.e. a complete metric space. Let $(\pi \in)\Pi = \mathscr{P}_{fin}(Id)$. We use the notation

$$\langle\!| \mathbf{X} |\!\rangle \overset{\text{not.}}{=} \Pi \times (Id \to \mathbf{X}).$$

Let θ ranges over $Id \to \mathbf{X}$. An element of type $\langle\!| \mathbf{X} |\!\rangle$ is a pair (π,θ), with $\pi \in \Pi$ and $\theta \in Id \to \mathbf{X}$.
We define mappings $id : \langle\!| \mathbf{X} |\!\rangle \to \Pi$, $(\cdot)(\cdot) : (\langle\!| \mathbf{X} |\!\rangle \times Id) \to \mathbf{X}$, and $\langle \cdot \mid \cdot \mapsto \cdot \rangle : (\langle\!| \mathbf{X} |\!\rangle \times Id \times \mathbf{X}) \to \langle\!| \mathbf{X} |\!\rangle$ by:

$$\begin{aligned} id(\pi,\theta) &= \pi,\\ (\pi,\theta)(\alpha) &= \theta(\alpha),\\ \langle(\pi,\theta) \mid \alpha \mapsto x\rangle &= (\pi \cup \{\alpha\}, \langle\theta \mid \alpha \mapsto x\rangle).\end{aligned}$$

Remark 10 *When X is a plain set (rather than a metric domain), we use the same notation $\langle\!| X |\!\rangle = \Pi \times (Id \to X)$ (with operators id, $(\cdot)(\cdot)$, $\langle \cdot \mid \cdot \mapsto \cdot\rangle$); only in this case $\langle\!| X |\!\rangle$ is not equipped with a metric.*

Notation 4 *The semantic models given in this paper are presented using a notation for* finite tuples *(*lists *or* sequences*) similar to the functional programming language Haskell (`www.haskell.org`) notation for lists; we use round brackets (rather than square brackets, that we use to represent multisets) to enclose the elements of a tuple. The elements in a list are separated by commas, and we use the symbol ':' as the cons operation. For example, the empty tuple is written as $()$, and if $(e \in)S$ is a set, $e_1,e_2,e_3 \in S$, then $(e_1,e_2,e_3) \in S^3$ ($S^3 = S \times S \times S$) and $(e_1,e_2,e_3) = e_1 : (e_2,e_3) = e_1 : e_2 : (e_3) = e_1 : e_2 : e_3 : ()$. By convention, $S^0 = \{()\}$. This notation is also used for metric domains [2]). For S a set, we denote by S^* the set of all finite (possibly empty) sequences over S.*

3 Continuation semantics for CCS^n

We started the semantic investigation of CCS^n with the language $\mathscr{L}_{syn}$ given in [2], based on CCS" (we only omit the CCS relabelling operator [15] which is not included in neither CCS^n nor CCS^{n+} [13]). In this section, we consider a language $\mathscr{L}_{CCS^n}$ which extends $\mathscr{L}_{syn}$ with the *joint input* construct of CCS^n

[9] $g(\alpha_1,\alpha_2) \leq \alpha_1$, $g(\alpha_1,\alpha_2) \leq \alpha_2$, and if $\alpha \leq \alpha_1$ and $\alpha \leq \alpha_2$, then $\alpha \leq g(\alpha_1,\alpha_2)$ for any $\alpha_1,\alpha_2,\alpha \in Id$.

and with the process algebra operators *left merge* $\lfloor\!\lfloor$, *synchronization merge* $\mid$ and *left synchronization merge* $\lfloor$. We refer the reader to [8, 9, 22] for further explanations regarding these auxiliary operators, which are essentially needed to make any element of the weakly abstract domain definable [4]. We use a set $(b \in)IAct$ of *internal actions* which contains a distinguished *silent action* τ, $\tau \in IAct$. We also use the set of *names* $(c \in)\mathcal{N}$ (see Definition 8) and a corresponding set of *co-names* $(\bar{c} \in)\overline{\mathcal{N}} = \{\bar{c} \mid c \in \mathcal{N}\}$. It is assumed that sets $IAct$ and $\mathcal{N} \cup \overline{\mathcal{N}}$ are disjoint: $IAct \cap (\mathcal{N} \cup \overline{\mathcal{N}}) = \emptyset$. The approach to recursion is based on *declarations* and *guarded statements* [2], and we use a set $(y \in)Y$ of *procedure variables*. Following [2], we work (without loss of generality) with a fixed declaration $D \in Decl$, and in any context we refer to such a fixed declaration D.

Definition 9 *The syntax of $\mathcal{L}_{CCS^n}$ is given by the following constructs:*

(a) Joint inputs $(j \in)J^n \quad j ::= c \mid j \,\&\, j$

(b) Elementary actions $(a \in)Act \quad a ::= b \mid \bar{c} \mid j \mid \mathsf{stop}$

(c) Statements $(x \in)Stmt, \quad x ::= a \mid y \mid x\backslash c \mid x;x \mid x+x \mid x \parallel x \mid x \mid x \mid x \lfloor\!\lfloor x \mid x \lfloor x$

(d) Guarded statements $(g \in)GStmt, \quad g ::= a \mid g\backslash c \mid g;x \mid g+g \mid g \parallel g \mid g \mid g \mid g \lfloor\!\lfloor x \mid g \lfloor g$

(e) Declarations $(D \in)Decl = Y \to GStmt$

(f) Programs $(\rho \in)\mathcal{L}_{CCS^n} = Decl \times Stmt$.

The class of elementary actions $(a \in)Act$ comprises elements of the following types: internal actions $b \in IAct$, output actions $\bar{c} \in \overline{\mathcal{N}}$, *joint inputs* $j \in J^n$ and the action stop which denotes *deadlock*. In addition, $\mathcal{L}_{CCS^n}$ provides operators for *sequential composition* $(x;x)$, *nondeterministic choice* $(x+x)$, *restriction* $x\backslash c$, *parallel composition* or *merge* $(x \parallel x)$, *left merge* $(x \lfloor\!\lfloor x)$, *synchronization merge* $(x \mid x)$, and *left synchronization merge* $(x \lfloor x)$. These operators are known from the classic process algebra theories. For instance, the restriction operator $x\backslash c$ is used to make the name c private within the scope of x [15].

Remark 11 *In $\mathcal{L}_{CCS^n}$, a* joint input *j is a construct $j = c_1 \,\&\, \cdots \,\&\, c_m$, where $1 \le m \le n$.*[10] *The language $\mathcal{L}_{CCS^{n+}}$ studied in Section 4 provides a more general construct $j = l_1 \,\&\, \cdots \,\&\, l_m$ called* joint prefix*, with $1 \le m \le n$, where $l_1, \ldots, l_m \in SAct$ are synchronization actions $SAct = \mathcal{N} \cup \overline{\mathcal{N}}$. For the remainder of this work, we assume a fixed positive natural number $\bar{n} \in \mathbb{N}^+$, such that at most $\bar{n}+1$ concurrent components can be involved in any (multiparty) interaction. However, note that $\bar{n}$ is a parameter of our formal specifications, and can be chosen to be arbitrarily large. For the language $\mathcal{L}_{CCS^n}$ we put $\bar{n} = n$ (n is the same number that occurs in the name of calculus CCS^n, in the name of language $\mathcal{L}_{CCS^n}$ and in the name of class J^n). In $\mathcal{L}_{CCS^n}$ (as in CCS^n [13]) $m+1$ actions $c_1 \,\&\, \cdots \,\&\, c_m$, $\bar{c}_1, \ldots, \bar{c}_m$, executed by $m+1$ concurrent processes can synchronize and their interaction is seen abstractly as a silent action τ.*

Definition 10 *In inductive reasoning, we use a complexity measure $wgt : Stmt \to \mathbb{N}$ defined as in [2]: $wgt(a) = 1$, $wgt(y) = 1 + wgt(D(y))$, $wgt(x\backslash c) = 1 + wgt(x)$, $wgt(x_1;x_2) = wgt(x_1 \lfloor\!\lfloor x_2) = 1 + wgt(x_1)$, and $wgt(x_1 \;\mathsf{op}\; x_2) = 1 + \max\{wgt(x_1), wgt(x_2)\}$, for $\mathsf{op} \in \{+, \parallel, \mid, \lfloor\}$.*

Definition 11 *(Interaction function for $\mathcal{L}_{CCS^n}$) Let $(u \in)U = \{u \mid u \in \mathcal{P}_{fin}(Act \times Id), |u| \le \bar{n}+1\}$ be the class of* interaction sets. *We write a pair $(a,\alpha) \in Act \times Id$ as $a@\alpha$. We define the* interaction function *$\iota : U \to (IAct \cup \{\uparrow\})$ (where $\uparrow \notin IAct$) by: $\iota(\{b@\alpha\}) = b$, $\iota(\{\mathsf{stop}@\alpha\}) = \iota(\{\bar{c}@\alpha\}) = \iota(\{j@\alpha\}) = \uparrow$, $\iota(\{\bar{c}_1@\alpha_1, \ldots, \bar{c}_m@\alpha_m, c_1 \,\&\, \cdots \,\&\, c_m@\alpha\}) = \mathsf{if}\; \iota_2(c_1@\alpha_1, c_1@\alpha) \wedge \cdots \wedge \iota_2(c_m@\alpha_m, c_m@\alpha) \;\mathsf{then}\; \tau \;\mathsf{else}\; \uparrow$, and $\iota(u) = \uparrow$ otherwise. The actions in a set $u \in U$ can interact iff $\iota(u) \in IAct$.*

[10] A joint input is written as $[c_1, \ldots, c_m]$ in CCS^n [13]. Since in this paper we use the notation based on square brackets $[\cdots]$ to represent *multisets* (as in [7], see Section 2), for a joint input we use the notation $c_1 \,\&\, \cdots \,\&\, c_m$.

3.1 Final Semantic Domains

We employ (linear time) metric domains $(q \in)\mathbf{Q}_D$ and $(q \in)\mathbf{Q}_O$ defined by domain equations [2]:

$$\begin{array}{rcl}\mathbf{Q}_D & \cong & \{\varepsilon\} + (IAct \times \frac{1}{2} \cdot \mathbf{Q}_D),\\ \mathbf{Q}_O & \cong & \{\varepsilon\} + \{\delta\} + (IAct \times \frac{1}{2} \cdot \mathbf{Q}_O),\end{array}$$

where ε is the *empty sequence* and δ models *deadlock*. The elements of $\mathbf{Q}_D$ and $\mathbf{Q}_O$ are finite or infinite sequences over $IAct$, and finite $\mathbf{Q}_O$ sequences can be terminated with δ. Instead of $(b_1,(b_2,\ldots,(b_n,\varepsilon)\ldots))$, $(b_1,(b_2,\ldots,(b_n,\delta)\ldots))$ and $(b_1,(b_2,\ldots))$, we write $b_1b_2\cdots b_n$, $b_1b_2\cdots b_n\delta$ and $b_1b_2\cdots$, respectively.

The metric domain $\mathbf{P}_D = \mathscr{P}_{nco}(\mathbf{Q}_D)$ is used as final domain for the denotational models presented in this paper. For the operational semantics, we use the metric domain $\mathbf{P}_O = \mathscr{P}_{nco}(\mathbf{Q}_O)$. The elements of $\mathbf{P}_O$ and $\mathbf{P}_D$ are nonempty and compact subsets of $\mathbf{Q}_D$ and $\mathbf{Q}_O$, respectively. For any $b \in IAct, q \in \mathbf{Q}_O$ $(q \in \mathbf{Q}_D)$ and $p \in \mathbf{P}_O$ $(p \in \mathbf{P}_D)$, we use the notations $b \cdot q = (x,q)$ and $b \cdot p = \{b \cdot q \mid q \in p\}$.

By τ^i and $\tau^i \cdot q$ we represent $\mathbf{Q}_D$ sequences defined inductively as follows: $\tau^0 = \varepsilon, \tau^0 \cdot q = q$ and $\tau^{i+1} = \tau \cdot \tau^i, \tau^{i+1} \cdot q = \tau \cdot (\tau^i \cdot q)$, for any $q \in \mathbf{Q}_D$ and $i \geq 0$. Also, for any $p \in \mathbf{P}_D$ and $i \geq 0$, we put $\tau^i \cdot p = \{\tau^i \cdot q \mid q \in p\}$.

Silent steps are needed to establish the contractiveness of function Ψ given in Definition 17. Following [9, 10], in the denotational model we use a sequence of the form $\tau^{\overline{n}}b$ to represent a successful interaction (among at most $\overline{n}+1$ concurrent processes), namely $\overline{n}$ silent steps $\tau^{\overline{n}}$ followed by an internal action b, which describes the effect of the interaction. Deadlock is modelled in the denotational model by a sequence of $\overline{n}$ silent steps $\tau^{\overline{n}}$ (not followed by an internal action).

Definition 12 *For any* $0 \leq i \leq \overline{n}$, *we define operators* $\oplus^i : (\mathbf{P}_D \times \mathbf{P}_D) \xrightarrow{1} \mathbf{P}_D$ *given by:*

$$p_1 \oplus^i p_2 = \ \mathsf{let}\ \overline{p} = \{q \mid q \in p_1 \cup p_2, q = \tau^{\overline{n}-i} \cdot \overline{q}, \overline{q} \neq \varepsilon\}\ \mathsf{in}\ \ \mathsf{if}\ \overline{p} = \emptyset\ \mathsf{then}\ \{\tau^{\overline{n}-i}\}\ \mathsf{else}\ \overline{p}.$$

The operators $\oplus^i$ *are well-defined, associative and commutative [9, 10].*

3.2 Operational semantics

The operational semantics of $\mathscr{L}_{CCS^n}$ is defined in the style of [18]. Following [2], we use the term *resumption* as an operational counterpart of the term *continuation*.

Definition 13 *(Resumptions and configurations) Let* $(f \in)SRes = \bigcup_{0 \leq i \leq \overline{n}} Stmt^i$ *be the class of* synchronous resumptions, *where* $Stmt^i = Stmt \times \cdots \times Stmt$ *(i times). Let us consider* $(r \in)R ::= E \bigm| x$, *where* $x \in Stmt$ *is a statement (Definition 9), and E is a symbol denoting* termination. *Also,* $(k \in)KRes = \langle\!|R|\!\rangle$ *(here* $\langle\!|R|\!\rangle$ *introduces a plain set, see Remark 10), and* $(\mu \in)Ids = \bigcup_{1 \leq i \leq \overline{n}+1} Id^i$ $(Id^i = Id \times \cdots Id$ *(i times). An element of the type Ids is a nonempty sequence of identifiers of length at most* $\overline{n}+1$. *Let* $ARes = Ids \times KRes$ *be the class of* asynchronous resumptions. *We write a pair* $(\mu,k) \in ARes$ *as* $\mu \cdot k$. *Let* $\alpha_0 \in Id$, $\alpha_0 = \bullet$, *and* $k_0 = (\emptyset, \lambda\alpha.E)$. *We define the class of* resumptions $(\rho \in)Res$ *as the smallest subset of* $(SRes \times U \times ARes)$, $Res \subseteq (SRes \times U \times ARes)$ *(where* $(u \in)U$ *is the set of interaction sets presented in Definition 11) satisfying the following axioms and rules:*

$$((),\emptyset,(\alpha_0)\cdot k_0) \in Res \qquad \frac{(x : f, u, \alpha : \mu \cdot k) \in Res \quad a \in Act \quad |u| \leq \overline{n}}{(f, \{a@\alpha\} \cup u, \mu \cdot k) \in Res}$$

$$\frac{(f,u,\alpha : \mu \cdot k) \in Res \quad c \in \mathscr{N}}{(f,u,\alpha(\bullet\backslash c) : \mu \cdot k) \in Res} \qquad \frac{(f,u,\alpha : \mu \cdot k) \in Res \quad x \in Stmt}{(f,u,\alpha(\bullet;) : \mu \cdot \langle k \mid \alpha(;\bullet) \mapsto x\rangle) \in Res}$$

$$\frac{(f,u,\alpha : \mu \cdot k) \in Res \quad x \in Stmt}{(f,u,\alpha(\bullet \parallel) : \mu \cdot \langle k \mid \alpha(\parallel \bullet) \mapsto x\rangle) \in Res} \qquad \frac{(f,u,\alpha : \mu \cdot k) \in Res \quad x \in Stmt \quad (len(f) + |u|) < \overline{n}}{(x : f,u,\alpha(\bullet \parallel) : \alpha(\parallel \bullet) : \mu \cdot k) \in Res}$$

where $len(f)$ *is the length of sequence* $f \in SRes$, *and* $|u|$ *is the cardinal number of the u.*

We also define the class of configurations $(t \in)Conf$ *by* $Conf = (Stmt \times Res) \cup R$.

Remark 12 *If we endow the set $\mathbb{N}\times\mathbb{N}$ with the* lexicographic ordering *denoted by $\prec$, we can define the complexity measure $c_{Res} : Res \to (\mathbb{N}\times\mathbb{N})$ by $c_{Res}(f,u,\mu\cdot k) = (|u|, c_\mu(\mu))$, where $|u|$ is the cardinal number of u, and for $\mu = (\alpha_1,\ldots,\alpha_m) \in Ids$ the mapping $c_\mu(\mu)$ is given by $c_\mu(\mu) = \sum_{1\le i\le m} c_\alpha(\alpha_i)$; here, $c_\alpha(\alpha)$ is the size of the term α (i.e., the number of nodes in the abstract syntax tree of α, $c_\alpha : Id \to \mathbb{N}$). One can verify that for any rule $\frac{\rho}{\rho'}$ presented in Definition 13 , we have $c_{Res}(\rho) \prec c_{Res}(\rho')$. Thus, any derivation tree proving that $\rho \in Res$ is finite.*

Before introducing the transition relation for $\mathscr{L}_{CCS^n}$, we present a mapping $ks : (Id \times KRes) \to R$ that is used to transform an element k of type $KRes$ into a value of type R:

$$\begin{aligned}
ks(\alpha,k) = {} & \mathsf{let}\ \pi = id(k)\ \mathsf{in}\\
& \mathsf{if}\ \pi|_\alpha = \emptyset\ \mathsf{then}\ E\ \ \mathsf{else}\ \ \mathsf{if}\ \pi|_\alpha = \{\alpha\}\ \mathsf{then}\ k(\alpha)\\
& \mathsf{else}\ \ \mathsf{if}\ \pi|_\alpha^{\mathscr{N}} = \{c\}\ \mathsf{then}\ ks(\alpha(\bullet\backslash c),k)\ \backslash^R\ c\\
& \mathsf{else}\ \ \mathsf{if}\ \alpha(;\bullet)\in\pi\ \mathsf{then}\ (ks(\alpha(\bullet;),k))\ ;^R (k(\alpha(;\bullet)))\ \mathsf{else}\ (ks(\alpha(\bullet\,\|),k))\ \|^R (ks(\alpha(\|\,\bullet),k)),
\end{aligned}$$

where the operators $\backslash^R : (R\times\mathscr{N}) \to R$, $;^R$, $\|^R : (R\times R)\to R$ are given by: $E \backslash^R c = E$, $x \backslash^R c = x\backslash c$, $E ;^R E = E$, $E ;^R x = x$, $x ;^R E = x$, $x_1 ;^R x_2 = x_1 ; x_2$, $E \ \|^R E = E$, $E\ \|^R x = x$, $x\ \|^R E = x$, $x_1\ \|^R x_2 = x_1 \parallel x_2$. For any $\pi\in\Pi$ and $\alpha\in Id$ we use the notation $\pi|_\alpha^{\mathscr{N}}$ given by:

$$\pi|_\alpha^{\mathscr{N}} = \{c \mid \alpha'\in\pi, match_\alpha^{\mathscr{N}}(\alpha,\alpha') = c \in \mathscr{N}\}$$

$$match_\alpha^{\mathscr{N}}(\bullet,(\alpha_2\backslash c)) = c,\quad match_\alpha^{\mathscr{N}}((\alpha_1\backslash c),(\alpha_2\backslash c)) = match_\alpha^{\mathscr{N}}(\alpha_1,\alpha_2),$$

$$match_\alpha^{\mathscr{N}}((\alpha_1;),(\alpha_2;)) = match_\alpha^{\mathscr{N}}((\alpha_1\,\|),(\alpha_2\,\|)) = match_\alpha^{\mathscr{N}}((\|\,\alpha_1),(\|\,\alpha_2)) = match_\alpha^{\mathscr{N}}(\alpha_1,\alpha_2),$$

and $\quad match_\alpha(\alpha_1,\alpha_2) =\uparrow\quad$ otherwise.

The type of mapping $match_\alpha^{\mathscr{N}}$ is $match_\alpha^{\mathscr{N}} : (Id\times Id) \to (\mathscr{N}\cup\{\uparrow\})$, with $\uparrow\notin\mathscr{N}$.

The transition relation $\to$ for language $\mathscr{L}_{CCS^n}$ is presented below by using the notation $t \xrightarrow{b} t'$ to expresses that $(t,b,t') \in\to$. Like in [2], in Definition 14 we write $t_1 \nearrow t_2$ as an abbreviation for $\dfrac{t_2 \xrightarrow{b} t'}{t_1 \xrightarrow{b} t'}$.

Definition 14 *The transition relation $\to$ for $\mathscr{L}_{CCS^n}$ is the smallest subset of $Conf \times IAct \times Conf$ satisfying the rules given below.*

(A0) $(a,((),u,(\alpha)\cdot k)) \xrightarrow{b} r$ $\qquad$ if $|u|\le\bar{n}, \iota(\{a@\alpha\}\cup u) = b, ks(\bullet,k) = r$

(R1) $(a,(x:f,u,\alpha:\mu\cdot k)) \nearrow (x,(f,\{a@\alpha\}\cup u,\mu\cdot k))$ $\quad$ if $|u|\le\bar{n}$

(R2) $(y,(f,u,\alpha:\mu\cdot k)) \nearrow (D(y),(f,u,\alpha:\mu\cdot k))$

(R3) $(x\backslash c,(f,u,\alpha:\mu\cdot k)) \nearrow (x,(f,u,\alpha(\bullet\backslash c):\mu\cdot k))$

(R4) $(x_1;x_2,(f,u,\alpha:\mu\cdot k)) \nearrow (x_1,(f,u,\alpha(\bullet;):\mu\cdot\langle k\mid\alpha(;\bullet)\mapsto x_2\rangle))$

(R5) $(x_1+x_2,(f,u,\alpha:\mu\cdot k)) \nearrow (x_1,(f,u,\alpha:\mu\cdot k))$

(R6) $(x_1+x_2,(f,u,\alpha:\mu\cdot k)) \nearrow (x_2,(f,u,\alpha:\mu\cdot k))$

(R7) $(x_1 \,\|\!\lfloor\, x_2,(f,u,\alpha:\mu\cdot k)) \nearrow (x_1,(f,u,\alpha(\bullet\,\|):\mu\cdot\langle k\mid\alpha(\|\,\bullet)\mapsto x_2\rangle))$

(R8) $(x_1 \lfloor x_2,(f,u,\alpha:\mu\cdot k)) \nearrow (x_1,(x_2:f,u,\alpha(\bullet\,\|):\alpha(\|\,\bullet):\mu\cdot k))$ $\qquad$ if $(len(f)+|u|)<\bar{n}$

(R9) $(x_1\,|\,x_2,(f,u,\alpha:\mu\cdot k)) \nearrow (x_1,(x_2:f,u,\alpha(\bullet\,\|):\alpha(\|\,\bullet):\mu\cdot k))$ $\qquad$ if $(len(f)+|u|)<\bar{n}$

(R10) $(x_1\,|\,x_2,(f,u,\alpha:\mu\cdot k)) \nearrow (x_2,(x_1:f,u,\alpha(\bullet\,\|):\alpha(\|\,\bullet):\mu\cdot k))$ $\qquad$ if $(len(f)+|u|)<\bar{n}$

(R11) $(x_1 \parallel x_2, (f,u,\alpha:\mu\cdot k)) \nearrow (x_1, (f,u,\alpha(\bullet\parallel):\mu\cdot\langle k \mid \alpha(\parallel\bullet)\mapsto x_2\rangle))$

(R12) $(x_1 \parallel x_2, (f,u,\alpha:\mu\cdot k)) \nearrow (x_2, (f,u,\alpha(\bullet\parallel):\mu\cdot\langle k \mid \alpha(\parallel\bullet)\mapsto x_1\rangle))$

(R13) $(x_1 \parallel x_2, (f,u,\alpha:\mu\cdot k)) \nearrow (x_1, (x_2:f,u,\alpha(\bullet\parallel):\alpha(\parallel\bullet):\mu\cdot k))$ if $(len(f)+|u|)<\overline{n}$

(R14) $(x_1 \parallel x_2, (f,u,\alpha:\mu\cdot k)) \nearrow (x_2, (x_1:f,u,\alpha(\bullet\parallel):\alpha(\parallel\bullet):\mu\cdot k))$ if $(len(f)+|u|)<\overline{n}$

(R15) $x \nearrow (x, ((),\emptyset,(\alpha_0)\cdot k_0))$.

In a configuration $(x,(f,u,\alpha:\mu\cdot k))$, α is the identifier of the *active computation* x, and the elements contained in μ are identifiers of the computations contained in the synchronous resumption f. Hence,we often represent a list of type *Ids* by highlighting the first element as $\alpha:\mu$, where μ can be the empty list. To define the behaviour of a restriction operation $x\backslash c$ evaluated in a context given by identifier α, in rule (R3) a new (local) context is created indicated by the identifier $\alpha(\bullet\backslash c)$ for the evaluation of statement x. To model multiparty interactions, joint inputs and output actions are added to the interaction set u, and an inference starts according to rule (R1) searching for a set of actions that could possibly interact. Axiom (A0) models the transition performed when it is found a set of elementary statements that can interact.

Definition 15 *For $t\in Conf$, we write $t \not\longrightarrow$ to express that there are no b,t' such that $t \xrightarrow{b} t'$. We say that t* terminates *if $t=E$, and that t* blocks *if $t\not\longrightarrow$ and t does not terminate.*

Definition 16 *(Operational semantics $\mathscr{O}[\![\cdot]\!]$ for $\mathscr{L}_{CCS^n}$) Let $(S\in)Sem_O = Conf \to \mathbf{P}_O$ ($\mathbf{P}_O$ was defined in Section 3.1). We define the higher order mapping $\Omega: Sem_O \to Sem_O$ by:*

$$\Omega(S)(t) = \begin{cases} \{\varepsilon\} & \text{if } t \text{ terminates} \\ \{\delta\} & \text{if } t \text{ blocks} \\ \bigcup\{b\cdot S(t') \mid t \xrightarrow{b} t'\} & \text{otherwise .} \end{cases}$$

We put $\mathscr{O}=\mathsf{fix}(\Omega)$. We also define $\mathscr{O}[\![\cdot]\!]: Stmt \to \mathbf{P}_O$ by $\mathscr{O}[\![x]\!] = \mathscr{O}(x,((),\emptyset,(\alpha_0)\cdot k_0))$.

To justify Definition 16, we note that the mapping Ω is a contraction (it has a *unique* fixed point, according to Banach's Theorem).

Example 1 *Let $x_1,x_2,x_3\in Stmt$, $x_1=(b_1\parallel b_2);\mathsf{stop}$, $x_2=((((b_1;(c_1\&c_2))\parallel\overline{c}_1)\backslash c_1)\parallel\overline{c}_2);(b_2+b_3)$, and $x_3=(((c_1\&c_2)\parallel\overline{c}_1)\backslash c_1)\parallel\overline{c}_2$. Considering $\overline{n}=2$, in all the examples presented in this paper we have at most $3(=\overline{n}+1)$ concurrent components interacting in each computing step. We use the function $\mathscr{O}[\![\cdot]\!]$ to compute the operational semantics for each of the three $\mathscr{L}_{CCS^n}$ programs x_1,x_2 and x_3. One can check that: $\mathscr{O}[\![x_1]\!]=\{b_1b_2\delta,b_2b_1\delta\}$, $\mathscr{O}[\![x_2]\!]=\{b_1\tau b_2,b_1\tau b_3\}$, and $\mathscr{O}[\![x_3]\!]=\{\tau\}$.*

Implementation: The operational and denotational semantics presented in this paper are available at `http://ftp.utcluj.ro/pub/users/gc/eneia/from24` as executable semantic interpreters implemented in Haskell. All $\mathscr{L}_{CCS^n}$ and $\mathscr{L}_{CCS^{n+}}$ programs presented in this paper (Example 1, Example 2 and Example 3) can be tested by using these semantic interpreters.

3.3 Denotational semantics

We define a denotational semantics $[\![\cdot]\!]: Stmt \to \mathbf{D}$ for $\mathscr{L}_{CCS^n}$, where (domain $\mathbf{P}_D$ is given in Section 3.1):

$$\begin{array}{rcll} (\phi\in)\mathbf{D} & \cong & \mathbf{Cont} \xrightarrow{1} \mathbf{P}_D & \\ (\gamma\in)\mathbf{Cont} & = & \mathbf{Cont}_S \times U \times \mathbf{Cont}_A & (\textit{continuations}) \\ (\varphi\in)\mathbf{Cont}_S & = & \sum_{i=0}^{\overline{n}} \mathbf{Sem}^i & (\textit{synchronous continuations}) \\ \mathbf{Cont}_A & = & Ids \times \mathbf{K} & (\textit{asynchronous continuations}) \\ (\kappa\in)\mathbf{K} & = & \langle\!|\mathbf{Den}|\!\rangle & (\phi\in)\mathbf{Sem} = \frac{1}{2}\cdot\mathbf{D} \qquad (\phi\in)\mathbf{Den} = \{\phi_E\}+\frac{1}{2}\cdot\mathbf{D}. \end{array}$$

The domain equation is given by the isometry $\cong$ between complete metric spaces. All basic sets (Id, Π, U and Ids) are equipped with the discrete metric (which is an ultrametric). The construction $\llparenthesis \cdot \rrparenthesis$ is presented in Section 2.3. According to [1, 2], this domain equation has a solution which is *unique* (up to isometry) and the solutions for $\mathbf{D}$ and all other domains presented above are obtained as complete ultrametric spaces.

We use semantic operators for restriction $\setminus : (\mathbf{D} \times \mathscr{N}) \to \mathbf{D}$, sequential composition $; : (\mathbf{D} \times \mathbf{D}) \to \mathbf{D}$, nondeterministic choice $\oplus : (\mathbf{D} \times \mathbf{D}) \to \mathbf{D}$, parallel composition (or merge) $\| : (\mathbf{D} \times \mathbf{D}) \to \mathbf{D}$, left merge $\lfloor\!\lfloor : (\mathbf{D} \times \mathbf{D}) \to \mathbf{D}$, left synchronization merge $\lfloor : (\mathbf{D} \times \mathbf{D}) \to \mathbf{D}$ and synchronization merge $\mid : (\mathbf{D} \times \mathbf{D}) \to \mathbf{D}$, defined with the aid of operators on continuations $\tilde{\setminus} : (\mathbf{Cont} \times \mathscr{N}) \to \mathbf{Cont}$, $add_{;} : (\mathbf{D} \times \mathbf{Cont}) \to \mathbf{Cont}$, $add_{\lfloor\!\lfloor} : (\mathbf{D} \times \mathbf{Cont}) \to \mathbf{Cont}$ and $add_{\lfloor} : (\mathbf{D} \times \mathbf{Cont}) \to \mathbf{Cont}$ as follows:

$$\phi \setminus c = \lambda\gamma\,.\,\phi(\gamma \tilde{\setminus} c), \qquad \phi_1 ; \phi_2 = \lambda\gamma\,.\,\phi_1(add_{;}(\phi_2, \gamma)), \qquad \phi_1 \lfloor\!\lfloor \phi_2 = \lambda\gamma\,.\,\phi_1(add_{\lfloor\!\lfloor}(\phi_2, \gamma)),$$

$$\phi_1 \oplus \phi_2 = \lambda\gamma\,.\,\phi_1(\gamma) \oplus^i \phi_2(\gamma), \quad \text{where } i = card_u(\gamma) \quad \text{(operators } \oplus^i \text{ are presented in Section 3.1)},$$

$$\phi_1 \lfloor \phi_2 = \text{ if } card_\gamma(\gamma) \text{ then } \lambda\gamma\,.\,\phi_1(add_{\lfloor}(\phi_2, \gamma)) \text{ else } \{\tau^{\overline{n}-|u|}\}, \qquad \phi_1 \mid \phi_2 = \phi_1 \lfloor \phi_2 \oplus \phi_2 \lfloor \phi_1$$

$$\phi_1 \parallel \phi_2 = \phi_1 \lfloor\!\lfloor \phi_2 \oplus \phi_2 \lfloor\!\lfloor \phi_1 \oplus \phi_1 \mid \phi_2, \qquad (\varphi, u, \alpha : \mu \cdot \kappa) \tilde{\setminus} c = (\varphi, u, \alpha(\bullet \setminus c) : \mu \cdot \kappa),$$

$$add_{;}(\phi, (\varphi, u, \alpha : \mu \cdot \kappa)) = (\varphi, u, \alpha(\bullet ;) : \mu \cdot \langle \kappa \mid \alpha(; \bullet) \mapsto \phi \rangle),$$

$$add_{\lfloor\!\lfloor}(\phi, (\varphi, u, \alpha : \mu \cdot \kappa)) = (\varphi, u, \alpha(\bullet \parallel) : \mu \cdot \langle \kappa \mid \alpha(\parallel \bullet) \mapsto \phi \rangle), \text{ and}$$

$$add_{\lfloor}(\phi, (\varphi, u, \alpha : \mu \cdot \kappa)) = (\phi : \varphi, u, \alpha(\bullet \parallel) : \alpha(\parallel \bullet) : \mu \cdot \kappa).$$

The mapping $card_u : \mathbf{Cont} \to \mathbb{N}$ is defined by $card_u(\varphi, u, \alpha : \mu \cdot \kappa) = |u|$, and predicate $card_\gamma : \mathbf{Cont} \to Bool$ is given by $card_\gamma(\varphi, u, \alpha : \mu \cdot \kappa) = ((len(\varphi) + |u|) < \overline{n})$, where $len(\varphi)$ is the length of sequence φ. Since operators $\oplus^i$ are associative and commutative [9, 10], the operator $\oplus$ is also associative and commutative. The mapping $kd : (Id \times \mathbf{K}) \to \mathbf{Den}$ is the semantic counterpart of function ks given in Section 3.2.

$$\begin{aligned}
kd(\alpha, \kappa) = {} & \text{let } \pi = id(\kappa) \text{ in} \\
& \quad \text{if } \pi|_\alpha = \emptyset \text{ then } \phi_E \text{ else if } \pi|_\alpha = \{\alpha\} \text{ then } \kappa(\alpha) \\
& \quad \text{else if } \pi|_\alpha^{\mathscr{N}} = \{c\} \text{ then } kd(\alpha(\bullet \setminus c), \kappa) \mathbin{\widehat{\setminus}} c \\
& \quad \text{else if } \alpha(; \bullet) \in \pi \text{ then } (kd(\alpha(\bullet ;), \kappa)) \mathbin{\widehat{;}} (k(\alpha(; \bullet))) \text{ else } (kd(\alpha(\bullet \parallel), \kappa)) \mathbin{\widehat{\parallel}} (kd(\alpha(\parallel \bullet), \kappa)).
\end{aligned}$$

Here, the operators $\widehat{\setminus} : (\mathbf{Den} \times \mathscr{N}) \to \mathbf{Den}$, $\widehat{;}$, $\widehat{\parallel} : (\mathbf{Den} \times \mathbf{Den}) \to \mathbf{Den}$ are given by: $\phi_E \mathbin{\widehat{\setminus}} c = \phi_E$, $\phi \mathbin{\widehat{\setminus}} c = \phi \setminus c$, $\phi_E \mathbin{\widehat{;}} \phi_E = \phi_E$, $\phi_E \mathbin{\widehat{;}} \phi = \phi \mathbin{\widehat{;}} \phi_E = \phi$, $\phi_1 \mathbin{\widehat{;}} \phi_2 = \phi_1 ; \phi_2$, $\phi_E \mathbin{\widehat{\parallel}} \phi_E = \phi_E$, $\phi_E \mathbin{\widehat{\parallel}} \phi = \phi \mathbin{\widehat{\parallel}} \phi_E = \phi$, $\phi_1 \mathbin{\widehat{\parallel}} \phi_2 = \phi_1 \parallel \phi_2$ – the notation $\pi|_\alpha^{\mathscr{N}}$ is presented in Section 3.2.

Definition 17 *(Denotational semantics $[\![\cdot]\!]$) Let $op_A : Act \to \mathbf{Cont} \to \mathbf{P}_D$ be given by:*

$$\begin{aligned}
& op_A(a)((), u, (\alpha) \cdot \kappa) = \text{ if } |u| \leq \overline{n} \text{ then } (\text{if } (\iota(\{a@\alpha\} \cup u) = b \in IAct, kd(\bullet, \kappa) = \phi_E) \text{ then } \{\tau^{\overline{n}-|u|} \cdot b\} \\
& \qquad\qquad \text{else } (\text{if } (\iota(\{a@\alpha\} \cup u) = b \in IAct, kd(\bullet, \kappa) = \phi \in \mathbf{D}) \text{ then } \tau^{\overline{n}-|u|} \cdot b \cdot \phi(\gamma_0) \text{ else } \{\tau^{\overline{n}-|u|}\})) \\
& op_A(a)(\phi : \varphi, u, (\alpha) \cdot \kappa) = \text{ if } |u| \leq \overline{n} \text{ then } \tau \cdot \phi(\varphi, \{a@\alpha\} \cup u, \mu \cdot \kappa) \text{ else } \{\tau^{\overline{n}-|u|} \cdot b\},
\end{aligned}$$

where $\gamma_0 = ((), \emptyset, (\alpha_0) \cdot \lambda\alpha\,.\,\phi_E)$ and $\alpha_0 = \bullet$ (as in Definition 13).

For $(S \in)F_D = Stmt \to \mathbf{D}$, we define the higher-order mapping $\Psi : F_D \to F_D$ by

$$\begin{array}{rcl}
\Psi(S)(a) & = & \lambda\gamma . op_A(a)(\gamma) \\
\Psi(S)(y) & = & \Psi(S)(D(y)) \\
\Psi(S)(x\backslash c) & = & \Psi(S)(x)\backslash c \\
\Psi(S)(x_1;x_2) & = & \Psi(S)(x_1)\,;S(x_2) \\
\Psi(S)(x_1+x_2) & = & \Psi(S)(x_1)\oplus\Psi(S)(x_2) \\
\Psi(S)(x_1\,\|\,x_2) & = & (\Psi(S)(x_1)\,\|\!\lfloor\, S(x_2))\oplus(\Psi(S)(x_2)\,\|\!\lfloor\, S(x_1))\oplus \\
 & & (\Psi(S)(x_1)\,\lfloor\,\Psi(S)(x_2))\oplus(\Psi(S)(x_2)\,\lfloor\,\Psi(S)(x_1)) \\
\Psi(S)(x_1\,|\,x_2) & = & (\Psi(S)(x_1)\,\lfloor\,\Psi(S)(x_2))\oplus(\Psi(S)(x_2)\,\lfloor\,\Psi(S)(x_1)) \\
\Psi(S)(x_1\,\|\!\lfloor\, x_2) & = & \Psi(S)(x_1)\,\|\!\lfloor\, S(x_2) \\
\Psi(S)(x_1\,\lfloor\, x_2) & = & \Psi(S)(x_1)\,\lfloor\,\Psi(S)(x_2).
\end{array}$$

We consider $\mathscr{D} = \mathsf{fix}(\Psi)$, and define $\mathscr{D}[\![\cdot]\!] : Stmt \to \mathbf{P}_D$ by $\mathscr{D}[\![x]\!] = \mathscr{D}(x)(\gamma_0)$.

Definition 17 can be easily justified by the techniques used in standard metric semantics [2], based on the observation that the definition of mapping $\Psi(S)(x)$ is structured by induction on the complexity measure $wgt(x)$ presented in Definition 10).

Example 2 *Let $x_1, x_2, x_3 \in Stmt$ be as in Example 1. Considering $\overline{n} = 2$, one can check that: $\mathscr{D}[\![x_1]\!] = \{\tau^{\overline{n}}b_1\tau^{\overline{n}}b_2\tau^{\overline{n}}, \tau^{\overline{n}}b_2\tau^{\overline{n}}b_1\tau^{\overline{n}}\}$, $\mathscr{D}[\![x_2]\!] = \{\tau^{\overline{n}}b_1\tau^{\overline{n}}\tau\tau^{\overline{n}}b_2, \tau^{\overline{n}}b_1\tau^{\overline{n}}\tau\tau^{\overline{n}}b_3\}$, and $\mathscr{D}[\![x_3]\!] = \{\tau^{\overline{n}}\tau\}$. For each x_i ($i = 1,2,3$), we observe that the result of $\mathscr{O}[\![x_i]\!]$ (given in Example 1) can be obtained from the yield of $\mathscr{D}[\![x_i]\!]$ if we omit the interspersed sequences $\tau^{\overline{n}}$ and replace a terminating sequence $\tau^{\overline{n}}$ with δ.*

4 Continuation semantics for CCS^{n+}

As explained in [13], the joint input construct of CCS^n "induces a unidirectional information flow". In [13], it is also studied a more general calculus called CCS^{n+} which can be obtained from CCS^n by replacing outputs and inputs with the *joint prefix* construct written as $[\alpha_1, \ldots, \alpha_m]$, where each α_i can be either an input action or an output action. Since we use the symbol α to represent identifiers, and the notation $[\ldots]$ to represent multisets, we employ a different notation. In this section, we consider a language named $\mathscr{L}_{CCS^{n+}}$ which can be obtained from $\mathscr{L}_{CCS^n}$ by replacing the output and joint input constructs with the joint prefix construct. We denote the $\mathscr{L}_{CCS^{n+}}$ *joint prefix* construct as $l_1 \,\&\, \cdots \,\&\, l_m$, where l is an element of the set of *synchronization actions SAct*, $(l \in)SAct = \mathscr{N} \cup \overline{\mathscr{N}}$. $(c \in)\mathscr{N}$ is the given set of *names* and $(\overline{c} \in)\overline{\mathscr{N}} = \{\overline{c} \mid c \in \mathscr{N}\}$ is the set of *co-names*. We use a mapping $\bar{\cdot} : SAct \to SAct$, defined such that $\overline{\overline{c}} = c$. The syntax of $\mathscr{L}_{CCS^{n+}}$ is similar to the syntax of $\mathscr{L}_{CCS^n}$. Only the classes of *joint prefixes* $(j \in)J_P^n$ and *elementary actions* $(a \in)Act$ are specific to $\mathscr{L}_{CCS^{n+}}$, and they are defined as follows:

$$j ::= l \mid j \,\&\, j \qquad\qquad a ::= b \mid j \mid \mathsf{stop}\ .$$

As in the case of language $\mathscr{L}_{CCS^n}$, b is an element of the class of *internal actions IAct* (which includes the distinguished element τ), and stop denotes *deadlock*. In $\mathscr{L}_{CCS^{n+}}$, the classes of *statements* $(x \in)Stmt$, *guarded statements* $(g \in)GStmt$ and *declarations* $(D \in)Decl$ remain as in Definition 9.

In $\mathscr{L}_{CCS^{n+}}$, a joint prefix is a construct $l_1 \,\&\, \cdots \,\&\, l_m$ with $1 \le m \le n$, where n is the number occurring in the name of the language $\mathscr{L}_{CCS^{n+}}$ and in the name of the syntactic class J_P^n. As in Section 4, we use the number $\overline{n}$ introduced in Remark 11 as a parameter of the formal specification of $\mathscr{L}_{CCS^{n+}}$. However, in the case of language $\mathscr{L}_{CCS^{n+}}$ we cannot simply put $\overline{n} = n$ (as we did for $\mathscr{L}_{CCS^n}$). The general rule is that $\overline{n}$ should be chosen sufficiently large such that at most $\overline{n}+1$ concurrent components are involved in a synchronous (multiparty) interaction in each computation step.

The flexibility provided by the technique of continuations (as a semantic tool) can handle a variety of complex interaction mechanisms with only minor changes to the formal specifications. Based on this flexibility, the continuation semantics for $\mathscr{L}_{CCS^{n+}}$ can be obtained easily from the semantic specification of $\mathscr{L}_{CCS^n}$. Only one modification in the semantic models of $\mathscr{L}_{CCS^n}$ is necessary to obtain the corresponding semantic models for $\mathscr{L}_{CCS^{n+}}$. Namely, we need to provide a new definition for the interaction function ι. The interaction function $\iota : U \to (IAct \cup \{\uparrow\})$ for language $\mathscr{L}_{CCS^{n+}}$ is defined by using $msync : U \to (IAct \cup \{\uparrow\})$ in the following way:

$$\iota(\{b@\alpha\}) = b, \quad \iota(\{\mathsf{stop}@\alpha\}) = \iota(\{j@\alpha\}) = \uparrow \quad \text{and} \quad \iota(u) = msync(u) \text{ otherwise, where}$$
$$msync(u) = \text{ if } u = \{j_1@\alpha_1,\ldots,j_m@\alpha_m\}, j_i \in J_P^n \quad \text{for all } i = 1,\ldots,n$$
$$\text{then let } w_r = [j_1@\alpha_1]^{rcv} \uplus \cdots \uplus [j_1@\alpha_1]^{rcv}, w_s = [j_1@\alpha_1]^{snd} \uplus \cdots \uplus [j_1@\alpha_1]^{snd}$$
$$\text{in if } (|w_r| = |w_s| \wedge \exists \varpi_r \in perm(w_r), \varpi_s \in perm(w_s)[match(\varpi_r, \varpi_s)]) \text{ then } \tau \text{ else } \uparrow$$
$$\text{else } \uparrow .$$

Neither in $\mathscr{L}_{CCS^n}$ nor in $\mathscr{L}_{CCS^{n+}}$ we impose the condition that the actions contained in a *joint input* or a *joint prefix* are distinct (these constructions describe multisets of actions). In the definition of *msync*, we let w range over the set $[SAct \times Id]$ of finite multisets of elements of type $SAct \times Id$, and $\uplus$ is the multiset sum operator (the notation for multisets is as in [7]). Also, we let ϖ range over the set $(SAct \times Id)^*$ of finite sequences over $SAct \times Id$; for the representation sequences we use lists (notation 4). We assume that *perm* is a function which computes the permutations of a multiset. If the mapping $msync(u)$ receives as argument a set $u = \{j_1@\alpha_1,\ldots,j_m@\alpha_m\}$ (where $j_i \in J_P^n$), then it splits the collection of actions contained in u into two multisets w_r and w_s containing input actions and output actions, respectively. For this purpose, it uses two mappings $[\cdot]^{rcv} : (J_P^n \times Id) \to [\mathscr{N} \times Id]$ and $[\cdot]^{snd} : (J_P^n \times Id) \to [\overline{\mathscr{N}} \times Id]$. For example, if $j = c_1 \& c_1 \& \bar{c}_2 \& \bar{c}_3$ then $[j@\alpha]^{rcv} = [c_1@\alpha, c_1@\alpha]$ and $[j@\alpha]^{snd} = [\bar{c}_2@\alpha, \bar{c}_3@\alpha]$. The function *msync* yields τ when it finds a pair of permutations of w_r and w_s that can match; it uses function *match*, which in turn uses the binary interaction mapping ι_2 presented in Section 2.3.

$$[c@\alpha]^{rcv} = [c@\alpha] \text{ if } c \in \mathscr{N}, [\bar{c}@\alpha]^{rcv} = [] \text{ if } \bar{c} \in \overline{\mathscr{N}}, \text{ and } [(j_1 \& j_2)@\alpha]^{rcv} = [j_1@\alpha]^{rcv} \uplus [j_2@\alpha]^{rcv}$$
$$[\bar{c}@\alpha]^{snd} = [\bar{c}@\alpha] \text{ if } \bar{c} \in \overline{\mathscr{N}}, [c@\alpha]^{snd} = [] \text{ if } c \in \mathscr{N}, \text{ and } [(j_1 \& j_2)@\alpha]^{snd} = [j_1@\alpha]^{snd} \uplus [j_2@\alpha]^{snd}$$
$$match((),()) = \mathsf{true}$$
$$match(c_r@\alpha_r : \varpi_r, c_s@\alpha_s : \varpi_s) = \text{ if } \iota_2(c_r@\alpha_r, \bar{c}_s@\alpha_s) \text{ then } match(\varpi_r, \varpi_s) \text{ else false}$$
$$\text{and} \quad match(\varpi_r, \varpi_s) = \mathsf{false} \quad \text{otherwise.}$$

Apart from this new definition of function ι, all other components of the formal specification of $\mathscr{L}_{CCS^{n+}}$ (including the semantic domains and all semantic operators) remain as in Section 3, for both the operational and the denotational semantics. Thus, we define the operational semantics $\mathscr{O}[\![\cdot]\!] : Stmt \to \mathbf{P}_O$ and the denotational semantics $\mathscr{D}[\![\cdot]\!] : Stmt \to \mathbf{P}_D$ for $\mathscr{L}_{CCS^{n+}}$ as in Definitions 16 and 17.

Example 3 *Let $x_4 \in Stmt$, $x_4 = ((((\bar{c}_1 \& c_2); b_1) \parallel (c_1 \& \bar{c}_3)) \backslash c_1 \parallel ((\bar{c}_2 \& c_3); b_2)$. This $\mathscr{L}_{CCS^{n+}}$ program is based on a CCS^{n+} example presented in [13]. Considering $\bar{n} = 2$, one can check that we have: $\mathscr{O}[\![x_4]\!] = \{\tau b_1 b_2, \tau b_2 b_1\}$, and $\mathscr{D}[\![x_4]\!] = \{\tau^{\bar{n}} \tau \tau^{\bar{n}} b_1 \tau^{\bar{n}} b_2, \tau^{\bar{n}} \tau \tau^{\bar{n}} b_2 \tau^{\bar{n}} b_1\}$.*

5 Weak abstractness of continuation semantics

Since the domain of CSC is not fully abstract [8], we study the abstractness of continuation semantics based on the weak abstractness principle presented in Section 2.2; thus, we show that the denotational models given in this article are weakly abstract with respect to their corresponding operational models. The proofs for $\mathscr{L}_{CCS^n}$ and $\mathscr{L}_{CCS^{n+}}$ are similar. Due to space limitations, we focus on the weak abstractness proof for $\mathscr{L}_{CCS^n}$, and show that the denotational semantics $\mathscr{D} : Stmt \to \mathbf{D}$ presented in Definition 17 is weakly abstract with respect to the operational semantics $\mathscr{O}[\![\cdot]\!] : Stmt \to \mathbf{P}_O$ presented in Definition 16.

We consider the class of *syntactic contexts* for $\mathscr{L}_{CCS^n}$ with typical element S (see Definition 18).

Definition 18 *(Syntactic contexts for $\mathscr{L}_{CCS^n}$)* $S ::= \circ \mid a \mid y \mid S\backslash c \mid S;S \mid S+S \mid S \parallel S \mid S|S \mid S \lfloor\!\lfloor S \mid S \lfloor S.$

For a context S and statement x, we denote by $S(x)$ the result of replacing all occurrences of $\circ$ in S with x.

Definition 19 *Let $\xi_Q : \mathbf{Q}_O \to \mathbf{Q}_D$ be the (unique) function [2] satisfying $\xi_Q(\varepsilon) = \varepsilon$, $\xi_Q(\delta) = \tau^{\overline{n}}$ and $\xi_Q(b \cdot q) = \tau^{\overline{n}} \cdot b \cdot \xi_Q(q)$. We also define $\xi_P : \mathbf{P}_O \to \mathbf{P}_D$ by $\xi_P(p) = \{\xi_Q(q) \mid q \in p\}$.*

We can now relate $\mathscr{D}[\![\cdot]\!]$ and $\mathscr{O}[\![\cdot]\!]$ for $\mathscr{L}_{CCS^n}$. The proof of Proposition 1 can proceed by using Lemma 5 and the observation that ξ_P is an injective function. We omit the proofs for Proposition 1 and Lemma 6; the reader can find similar results in [21].

Lemma 6 $\xi_P(\mathscr{O}[\![x]\!]) = \mathscr{D}[\![x]\!]$, *for all $x \in Stmt$.*

Proposition 1 *The denotational semantics $\mathscr{D}$ is* correct *with respect to the operational semantics $\mathscr{O}[\![\cdot]\!]$.*

Definition 20 *Let $[\![\cdot]\!]_K : Kres \to \mathbf{K}$ be given by $[\![k]\!]_K = (id(k), \lambda\alpha\,.\ \mathsf{if}\ k(\alpha) = E\ \mathsf{then}\ \phi_E\ \mathsf{else}\ \mathscr{D}(k(\alpha)))$. We define the mapping $[\![\cdot]\!]_F : SRes \to \mathbf{Cont}_S$ by $[\![()]\!]_F = ()$ and $[\![(x : f)]\!]_F = \mathscr{D}(x) : [\![f]\!]_F$. We also define the mapping $[\![\cdot]\!]_C : Res \to \mathbf{Cont}$ by $[\![(f,u,\alpha : \mu \cdot k)]\!]_C = ([\![f]\!]_F, u, \alpha : \mu \cdot [\![k]\!]_K)$, and consider $\mathbf{Cont}^{\mathscr{D}} = \{[\![(f,u,\alpha:\mu \cdot k)]\!]_C \,|\, (f,u,\alpha : \mu \cdot k) \in Res\}$. Clearly, $\mathbf{Cont}^{\mathscr{D}}$ is a subspace of $\mathbf{Cont}$, i.e. $\mathbf{Cont}^{\mathscr{D}} \lhd \mathbf{Cont}$.*

Lemma 7 $\mathbf{Cont}^{\mathscr{D}}$ *is a class of denotable continuations for the denotational semantics $\mathscr{D}$ of $\mathscr{L}_{CCS^n}$.*

Proof. Let $\mathbf{D}^{\mathscr{D}} = \{\mathscr{D}(s) \,|\, s \in Stmt\}$. We must show that $\mathbf{Cont}^{\mathscr{D}}$ is invariant for $\mathbf{D}^{\mathscr{D}}$ under all operators used in the definition of $\mathscr{D}$. We only handle operator $add_{\lfloor} : (\mathbf{D} \times \mathbf{Cont}) \to \mathbf{Cont}$ given in Section 3.3. The other operators can be handled similarly. The semantic operators used in the definition of a denotational semantics are nonexpansive; this means that we can write $add_{\lfloor}$ as $add_{\lfloor} : (\mathbf{D} \times \mathbf{Cont}) \xrightarrow{1} \mathbf{Cont}$, namely $add_{\lfloor} \in \mathscr{O}t$, where $\mathscr{O}t = (\mathbf{D} \times \mathbf{Cont}) \xrightarrow{1} \mathbf{Cont}$. We show now that $add_{\lfloor} \in \mathscr{O}t(\mathbf{D}^{\mathscr{D}}, \mathbf{Cont}^{\mathscr{D}})$, i.e., $add_{\lfloor}(\phi, \gamma) \in \mathbf{Cont}^{\mathscr{D}}$ for any $\phi \in \mathbf{D}^{\mathscr{D}}$, $\gamma \in \mathbf{Cont}^{\mathscr{D}}$. Since $\phi \in \mathbf{D}^{\mathscr{D}}$ and $\gamma \in \mathbf{Cont}^{\mathscr{D}}$, $\phi = \mathscr{D}(x)$ for some $x \in Stmt$, and $\gamma = [\![(f,u,\alpha : \mu \cdot k)]\!]_C = ([\![f]\!]_F, u, \alpha : \mu \cdot [\![k]\!]_K)$ for some $(f,u,\alpha : \mu \cdot k) \in Res$. Assuming $len(f) + |u| < \overline{n}$, we get $add_{\lfloor}(\phi,\gamma) = add_{\lfloor}(\mathscr{D}(x), ([\![f]\!]_F, u, \alpha : \mu \cdot [\![k]\!]_K)) = (\mathscr{D}(x) : [\![f]\!]_F, u, \alpha(\bullet \parallel) : \alpha(\parallel \bullet) : \mu \cdot [\![k]\!]_K))$ $= ([\![x : f]\!]_F, u, \alpha(\bullet \parallel) : \alpha(\parallel \bullet) : \mu \cdot [\![k]\!]_K) = [\![(x : f, u, \alpha(\bullet \parallel) : \alpha(\parallel \bullet) : \mu \cdot k)]\!]_C = \gamma'$. By Definition 13, since $(f,u,\alpha : \mu \cdot k) \in Res$, we also have $(x : f, u, \alpha(\bullet \parallel) : \alpha(\parallel \bullet) : \mu \cdot k) \in Res$. Thus, we infer that $\gamma' = [\![(x : f, u, \alpha(\bullet \parallel) : \alpha(\parallel \bullet) : \mu \cdot k)]\!]_C \in \mathbf{Cont}^{\mathscr{D}}$, as required. □

By Remark 9, the domain of denotable continuations $co(\mathbf{Cont}^{\mathscr{D}}|\mathbf{Cont})$ is invariant for $co(\mathbf{D}^{\mathscr{D}}|\mathbf{D})$ under all operators used in the definition of $\mathscr{D}$. The proof of Lemma 8 is by induction on the depth of the inference of $(f,u,\alpha : \mu \cdot k) \in Res$ by using the rules given in Definition 13.

Lemma 8 *For any $x \in Stmt$, $(f,u,\alpha : \mu \cdot k) \in Res$ there is an $\mathscr{L}_{CCS^n}$ syntactic context S such that $\tau^{|u|} \cdot \mathscr{D}(x)[\![(f,u,\alpha : \mu \cdot k)]\!]_C = \mathscr{D}[\![S(x)]\!] = \mathscr{D}(S(x))(\gamma_0)$. Furthermore, S does not depend on x; it only depends on $(f,u,\alpha : \mu \cdot k)$. For any $x, x' \in Stmt$, $x \neq x'$, $(f,u,\alpha : \mu \cdot k) \in Res$ there is a syntactic context S such that: $\tau^{|u|} \cdot \mathscr{D}(x)[\![(f,u,\alpha : \mu \cdot k)]\!]_C = \mathscr{D}[\![S(x)]\!]$ and $\tau^{|u|} \cdot \mathscr{D}(x')[\![(f,u,\alpha : \mu \cdot k)]\!]_C = \mathscr{D}[\![S(x')]\!]$ (the same S in both equalities). Notice that in general, $\mathscr{D}[\![S(x)]\!] \neq \mathscr{D}[\![S(x')]\!]$.*

Theorem 5 *The denotational semantics $\mathscr{D}$ of $\mathscr{L}_{CCS^n}$ is* weakly abstract *with respect to the operational semantics $\mathscr{O}[\![\cdot]\!]$.*

Proof. By Proposition 1, $\mathscr{D}$ is *correct* with respect to $\mathscr{O}[\![\cdot]\!]$. For the weak completeness condition, we consider the class of denotable continuations $\mathbf{Cont}^{\mathscr{D}}$ for $\mathscr{D}$ presented in Definition 20. By Lemma 5, if we prove that $\forall x_1, x_2 \in Stmt[(\exists \gamma \in \mathbf{Cont}^{\mathscr{D}}[\mathscr{D}(x_1)\gamma \neq \mathscr{D}(x_2)\gamma]) \Rightarrow (\exists S[\mathscr{O}(S(x_1)) \neq \mathscr{O}(S(x_2))])]$, then $\mathscr{D}$ is also *weakly complete* with respect to $\mathscr{O}[\![\cdot]\!]$. Let $x_1, x_2 \in Stmt$ and $(f, u, \alpha : \mu \cdot k) \in Res$ be such that $\mathscr{D}(x_1)[\![(f, u, \alpha : \mu \cdot k)]\!]_C \neq \mathscr{D}(x_2)[\![(f, u, \alpha : \mu \cdot k)]\!]_C$ which implies $\tau^{|u|} \cdot \mathscr{D}(x_1)[\![(f, u, \alpha : \mu \cdot k)]\!]_C \neq \tau^{|u|} \cdot \mathscr{D}(x_2)[\![(f, u, \alpha : \mu \cdot k)]\!]_C$. By Lemma 8, there is an $\mathscr{L}_{CCS^n}$ syntactic context S such that $\mathscr{D}[\![S(x_1)]\!] = \tau^{|u|} \cdot \mathscr{D}(x_1)[\![(f, u, \alpha : \mu \cdot k)]\!]_C \neq \tau^{|u|} \cdot \mathscr{D}(x_2)[\![(f, u, \alpha : \mu \cdot k)]\!]_C = \mathscr{D}[\![S(x_2)]\!]$. By using Lemma 6, we get $\xi_P(\mathscr{O}[\![S(x_1)]\!]) = \mathscr{D}[\![S(x_1)]\!] \neq \mathscr{D}[\![S(x_2)]\!] = \xi_P(\mathscr{O}[\![S(x_2)]\!])$ which implies $\mathscr{O}[\![S(x_1)]\!] \neq \mathscr{O}[\![S(x_2)]\!]$. Thus, we conclude that $\mathscr{D}$ is weakly complete, and therefore *weakly abstract* with respect to $\mathscr{O}[\![\cdot]\!]$. □

In a similar way, one can show that the denotational semantics of $\mathscr{L}_{CCS^{n+}}$ is weakly abstract with respect to the corresponding operational model.

6 Conclusion

While the classic full abstractness condition cannot be established in continuation semantics [4], the abstractness of a continuation-based denotational model can be investigated based on the weak abstractness criterion. Compared to the classic full abstractness criterion [14], the weak abstractness criterion used in this paper relies on a weaker completeness condition that should be verified only for a class of denotable continuations.

We provide the denotational and operational semantics defined by using continuations for two process calculi (based on CCS) able to express multiparty synchronous interactions. We work with metric semantics and with the continuation semantics for concurrency (a technique introduced by the authors to handle advanced concurrent control mechanisms). For the multiparty interaction mechanisms incorporated in both process calculi, we proved that the continuation-based denotational models are weakly abstract with respect to their corresponding operational models.

As future work, we intend to investigate the weak abstractness issue for nature-inspired approaches introduced in the area of membrane computing [17, 19, 10].

References

[1] P. America and J.J.M.M. Rutten, "Solving Reflexive Domain Equations in a Category of Complete Metric Spaces," *J. of Comp. Syst. Sci*, vol. 39, pp. 343–375, 1989. `https://doi.org/10.1016/0022-0000(89)90027-5`

[2] J.W. de Bakker and E.P. de Vink, *Control Flow Semantics,* MIT Press, 1996.

[3] J.W. de Bakker and J.I. Zucker, "Compactness in Semantics for Merge and Fair Merge," in E. Clarke and D. Kozen, editors, *Proc. of Workshop on Logics of Programs (LNCS, vol. 164)*, pp. 18—33, 1983. `https://doi.org/10.1007/3-540-12896-4_352`

[4] R. Cartwright, P.-L. Curien and M. Felleisen, "Fully Abstract Semantics for Observably Sequential Languages," *Information and Computation*, vol. 111, pp. 297–401, 1994. `https://doi.org/10.1006/inco.1994.1047`

[5] G. Ciobanu and E.N. Todoran, "Abstract continuation semantics for asynchronous concurrency," Technical Report FML-12-02, Romanian Academy, 2012. Available at `https://ftp.utcluj.ro/pub/users/gc/eneia/fml12/fml1202.pdf`

[6] G. Ciobanu and E.N. Todoran, "Continuation Semantics for Asynchronous Concurrency," *Fundamenta Informaticae*, vol. 131(3-4), pp. 373–388, 2014. `https://doi.org/10.3233/FI-2014-1020`

[7] G. Ciobanu and E.N. Todoran. "Correct metric semantics for a language inspired by DNA computing," Concurrency and Computation: Practice and Experience, vol. 28, pp. 3042–3060, 2016. `https://doi.org/10.1002/cpe.3585`

[8] G. Ciobanu and E.N. Todoran, "Abstract Continuation Semantics for Asynchronous Concurrency," *Proc. IEEE SYNASC 2017*, pp. 296–303. `https://doi.org/10.1109/SYNASC.2017.00056`

[9] G. Ciobanu, E.N. Todoran, "A Study of Multiparty Interactions in Continuation Semantics," Proc. IEEE SYNASC 2020, pp. 117–124, 2020. `https://doi.org/10.1109/SYNASC51798.2020.00029`

[10] G. Ciobanu and E.N. Todoran, "A process calculus for spiking neural P systems," *Information Sciences*, vol. 604, pp. 298–319, 2022. `https://doi.org/10.1016/j.ins.2022.03.096`

[11] R.K. Dybvig, R. Hieb. Engines from Continuations, *Computer Languages*, vol.14(2), 109–123, 1989. `https://doi.org/10.1016/0096-0551(89)90018-0`

[12] D.P. Friedman, C. T. Haynes, M. Wand. Obtaining Coroutines with Continuations, *Computer Languages*, vol.11(3/4), 143–153, 1986. `https://doi.org/10.1016/0096-0551(86)90007-X`

[13] C. Laneve and A. Vitale, "The Expressive Power of Synchronizations," Proc. of LICS 2010, pp. 382-–391, 2010. `https://doi.org/10.1109/LICS.2010.15`

[14] R. Milner, "Fully Abstract Models of Typed λ-Calculi," *Theoretical Computer Science*, vol. 4, pp. 1–22, 1977. `https://doi.org/10.1016/0304-3975(77)90053-6`

[15] R. Milner. *Communication and Concurrency*, Prentice Hall, 1989.

[16] P.D. Mosses, "Programming language description languages," in Formal Methods: State of the Art and New Directions, pp. 249–273, Springer, 2010. `https://doi.org/10.1007/978-1-84882-736-3_8`

[17] Gh. Păun, G. Rozenberg, and A. Salomaa, editors, The Oxford Handbook of Membrane Computing, Oxford University Press, 2010.

[18] G. Plotkin, "A Structural Approach to Operational Semantics," *J. Log. and Algebr. Program.*, vol. 60–61, pp. 17–139, 2004. `https://doi.org/10.1016/j.jlap.2004.03.009`

[19] B. Song, K. Li, D.Orellana-Martín, M.J. Pérez-Jiménez, Ignacio Pérez-Hurtado, "A Survey of Nature-Inspired Computing: Membrane Computing," *ACM Comput. Surv.*, vol. 54(1), pp. 22:1–22:31, 2021. `https://doi.org/10.1145/3431234`

[20] C. Strachey and C.P. Wadsworth, "Continuations: A Mathematical Semantics for Handling Full Jumps," *Higher-Order and Symbolic Computation*, vol. 13, pp. 135–152, 2000. `https://doi.org/10.1023/A:1010026413531`

[21] E.N. Todoran, "Metric Semantics for Synchronous and Asynchronous Communication: A Continuation-based Approach," *Electronic Notes in Theoretical Computer Science*, vol. 28, pp. 101–127, 2000. `https://doi.org/10.1016/S1571-0661(05)80632-2`

[22] E.N., Todoran, "Continuation-based metric semantics for concurrency," Proc. IEEE ICCP 2019, pp. 551–559, 2019. `https://doi.org/10.1109/ICCP48234.2019.8959761`

Towards Geometry-Preserving Reductions Between Constraint Satisfaction Problems (and other problems in NP)

Gabriel Istrate
University of Bucharest
Str. Academiei 14, Sector 6, 011014, Bucharest, Romania
gabriel.istrate@unibuc.ro

We define two kinds of *geometry-preserving reductions* between constraint satisfaction problems and other NP-search problems. We give a couple of examples and counterexamples for these reductions.

Keywords: solution geometry, overlaps, covers, reductions, computational complexity.

1 Introduction

Reductions are the fundamental tool of computational complexity. They allow classification of computational problems into families (somewhat resembling those in biology), with problems in the same class sharing common features (in particular NP-complete problems being isomorphic versions of a single problem, if we believe that the Berman-Hartmanis conjecture [6] holds).

A significant concern in the literature on complexity-theoretic reductions is to make the theory of reductions more predictive. This means that we should aim to better connect the structural properties of the two problems that the given reduction is connecting. For instance the notion of *pasimonious reduction* attempts to preserve the total number of solutions between instances. Various attempts (e.g. [12, 5]) have proposed restricting the computational power allowed for computing reductions. (Strongly) local reductions [23, 22] provide a principled approach that *simultaneously* relates the complexity of various versions (decision, counting,etc) of the two problems in the reduction. In another direction, several logic-based approaches to the complexity of reductions were considered [47, 13]. Finally, motivated by the connection with local search, a line of research [17], [18] has investigated reductions that attempt to preserve the *structure* of solution spaces, by not only translating the instances of the two decision problems but also the set of *solutions*.

The geometric structure of solution spaces of combinatorial problems has been brought to attention by an entirely different field of research: that of *phase transitions in combinatorial optimization* [44, 38]. It was shown that the predictions of the so-called *one-level replica symmetry breaking* in Statistical Physics applies to (and offer predictions for) the complexity of many constraint satisfaction problems. It is believed [32] that the set of solutions of a constraint satisfaction problem such as k-SAT undergoes a sequence of phase transitions, culminating in the SAT/UNSAT transition. The first transition is a *clustering transition* where the set of satisfying assignments of a typical random formula changes from a single, connected cluster to an exponential number of clusters that are far apart from eachother. Some aspects of this solution clustering have been confirmed rigorously (*solution clustering* [3, 39, 14, 28, 30, 4]). The 1-RSB prediction motivated the development of the celebrated *survey propagation (SP) algorithm* [7] that predicts (for large enough k) the location of the phase transition of the random k-SAT problem [15]. This breakthrough has inspired significant recent progress [10, 19, 20, 8].

A major weakness of the theory of phase transition is the lack of general results that connect the qualitative features such as clustering and 1-RSB to computational complexity. In contrast, Computational Complexity has developed algebraic classification tools that (at least for the well-behaved class of Constraint Satisfaction Problems) provide an understanding of the structural reasons for the (in)tractability of various CSP [46, 9, 53]. Note, however, that there are some classification results in the area of Phase Transitions: For instance, one can classify the existence of thresholds for random Constraint Satisfaction Problems in a manner somewhat reminiscent of the Schaefer Dichotomy Theorem [40, 21, 24, 11, 27]. Intuitively, SAT problems without a sharp threshold qualitatively resemble Horn SAT, a problem whose (coarse) threshold is well understood [25, 26, 42]. Also, versions of SAT that have a "second-order phase transition" [41] seem qualitatively similar to 2-SAT [29]. An example of such a problem which is NP-complete but "2-SAT-like" is 1-in-K SAT, $k \geq 3$ [1]. Finally, the existence of overlap discontinuities can be approached in a uniform manner for a large class of random CSP [28]. In spite of all this, the existence of general classification results does **not** extend to properties (such as 1-RSB) that concern the geometric nature of the solution space of random CSP.

In this paper **we propose connecting these two directions by the definition of reductions that take into account the geometric structure of solution spaces**. We put forward two such notions:

- the first type of reductions extends the witness-isomorphic reductions of [18] by requiring that the solution overlap of the target (harder) problem depends predictably on the overlap of the source (easier) problem. In other words, the cluster structure of the harder problem is essentially the same as the one of the easier problem.

- for a subclass of NP-search problems, that of constraint satisfaction problems, we define a type of witness-preserving reductions based on the notion of *covers*. Covers [35, 31] were defined in the course of analyzing the seminal *survey propagation* algorithm of Zecchina et al. [7]. They are generalizations of (sets of) satisfying assignments for the problem, and allow a geometric interpretation of this algorithm. Specifically, survey propagation is just belief propagation applied to the set of covers [35]. Our reduction extend the usual notion of reductions by requiring that not only solutions of one problem are mapped to solutions, but this is true *for covers as well*.

The outline of the paper is as follows: in Section 2 we subject the reader to a (rather heavy) barrage of definitions and concepts. The two kinds of reductions we propose are given in Definitions 12 and 13. In Section 3 we give four examples of problem reductions (of various naturalness and difficulty) that belong to at least one of the two classes of reductions that we put forward. The first two are classic examples from the literature on NP-completeness. The next two are slightly less trivial examples, and are constructed to be appropriate for one of the two types of reductions (one for each type). Correspondingly, in Section 4 we point out that the natural reduction from 4-SAT to 3-SAT belongs to neither class. An algebraic perspective on the theory of reductions is called for in Section 5. We end (Section 6) with a number of musings concerning the merits and pitfalls of our approach, and the road ahead.

We stress the fact that **this is primarily a conceptual paper**: its added value consists primarily, we believe, in pointing to the right direction in a problem where nailing the correct concepts is tricky, thus stimulating further discussion, rather than in coming up with completely appropriate concepts and proving things about them. **Our main goal is to understand and model, not to prove.** However, if "right" definitions exist, they should be compatible with our results.

2 Preliminaries

We review some notions that we will need in the sequel:

Definition 1. *An NP-search problem is specified by*

- *a polynomial time computable predicate $A \subseteq \Sigma^*$. A specifies the set of well-formed instances.*
- *a polynomial time computable relation $R \subseteq A \times \Sigma^*$ with the property that there exists a polynomial-time computable (polynomially bounded) function $q(\cdot)$ such that for all pairs $(x,y) \in A \times \Sigma^*$ such that $R(x,y)$ (y is called a witness for x, and x is called a positive instance), we have $|y| = q(|x|)$*[1]*.*
- *a polynomial time computable relation $N \subseteq A \times \Sigma^* \times \Sigma^*$ such that if $N(x,y_1,y_2)$ then $|y_1| = |y_2| = q(|x|)$. Intuitively, N encodes the fact that y_1, y_2 are "neighbors". Note that it is* ***not*** *required that $R(x,y_1), R(x,y_2)$, i.e. that y_1, y_2 are witnesses for x. Instead, y_1, y_2 are "candidate witnesses".*

We abuse language and write R instead of (A,R,N). We denote by NP_S the class of NP-search problems. A problem R in NP_S belongs to the class P_S iff there exists a polynomial time algorithm W s.t.

- *For all $x \in A$, if $W(x) =$ "NO" then for all $y \in \Sigma^*$, $(x,y) \notin R$.*
- *For all $x \in A$, if $W(x) = y \neq$ "NO" then $R(x,y)$.*

Definition 2. *Given NP-search problem R and positive instance x, define the overlap of two witnesses y_1, y_2 for x as*

$$overlap(y_1,y_2) = \frac{d_H(y_1,y_2)}{q(|x|)} \tag{1}$$

In the previous equation d_H denotes, of course, the Hamming distance of two strings.

We will employ a particular kind of NP-search problem that arises from optimization. The particular class of interest consists of those optimization problems for which determining if a candidate witness is a local optimum, and finding a better candidate witness otherwise, is tractable:

Definition 3. *An NP-search problem R belongs to the class PLS (polynomial local search, see e.g. [52]) if there exist three polynomial time algorithms A, B, C such that*

- *Given $x \in \Sigma^*$, A determines if x is a legal instance (as in Definition 1), and, additionally, if this is indeed the case, it produces some candidate witness $s_0 \in \Sigma^{q(|x|)}$.*
- *Given $x \in A$ and string s, B computes a cost $f(x,s) \in \mathbf{N} \cup \{\infty\}$*[2]*. s is called a local minimum if $f(x,s) \leq f(x,s')$ for all s' such that $N(x,s,s')$. Local maxima are defined similarly.*
- *Given $x \in A$ and string s, C determines whether s is a local optimum and, if it not, it outputs a string s' with $N(x,s,s')$ such that $f(x,s') < f(x,s)$ (if R is a minimization problem; the relation is reversed if R is a maximization problem).*

In addition we require that $R(x,s)$ (i.e. s is a witness for x) if and only if s is a local optimum for x. That is, witnesses for an instance x are local optima.

Observation 1. *It is known (see e.g. [52]) that $P_S \subseteq PLS \subseteq NP_S$.*

[1]normally we require only the inequality $|y| \leq q(|x|)$, but we pad witnesses y, if needed, to ensure that this condition holds.

[2]Rather than adding another relation to test whether s is a legal witness candidate for x, we chose to do this implicitly, by requiring that $f(x,s) < \infty$

Example 1. *MAX-SAT is the following optimization problem: an instance of MAX-SAT is a CNF formula Φ whose clauses $C_1, C_2, \dots, C_m$ are endowed with a positive weight $w_1, w_2, \dots, w_m$, respectively. The set of witnesses for Φ is the set of assignments of the variables in Φ. Each assignment A comes with a weight $w(A)$, defined to be the sum of weights of all clauses satisfied by A.*

A candidate witness for Φ is a truth assignment of the variables in Φ. Two candidate witnesses w_1, w_2 are neighbors if they differ on the value of a single variable.

A witness for Φ is an assignment $w(A)$ which is a local maximum, i.e for all assignments B differing from A in exactly one position $w(B) \leq w(A)$.

It is clear that (as defined) problem MAX-SAT belongs to class PLS.

Definition 4. *Given problem $A \in PLS$, define the NP-search problem A^* as follows:*

- *Inputs of A^* consist of pairs $(x, \tau, 0^d)$, where x is an input for A, τ is a string in Σ^* for some finite alphabet Σ, $|\tau| \leq q(|x|)$, and $d \geq 1$ is an integer.*
- *A witness for pair $(x, \tau, 0^d)$ is a path $\tau_0, \tau_1, \dots, \tau_{d'}$ with $d' \leq d$ such that (a). $\tau_0 = \tau$ (b). for every $i \geq 1$ τ_{i-1} and τ_i are neighbors and $f(x, \tau_{i-1}) < f(x, \tau_i)$. (c). $\tau_{d'}$ is a local optimum in A (such a path is called an augmenting path).*

In other words, given instance $(x, \tau, 0^d)$ the problem A^* is to decide whether there exists an augmenting path of length at most d from τ to a local optimum.

To define cover-preserving reductions we first need **witness-isomorphic reductions**, a very restrictive notion that has been previously studied in the literature [18].

Definition 5. *Given NP-decision problems (A, R_A) and (B, R_B) a witness-isomomorphic reduction of (A, R_A) to (B, R_B) is specified by two polynomial time computable and polynomial time invertible functions f, g with the following properties:*

- *for all x, $x \in A \Leftrightarrow f(x) \in B$. That is, f witnesses that $A \leq_m^P B$.*
- *for all x, y if $R_A(x, y)$ then there exists a z such that $g(<x, y>) = <f(x), z>$ and $R_B(f(x), z)$.*
- *for all distinct y_1, y_2 if $R(x, y_1)$ and $R(x, y_2)$ then $g(<x, y_1>) \neq g(<x, y_2>)$.*
- *for all x, w, z if $f(x) = w$ and $R_B(w, z)$ then there exists y such that $R_A(x, y)$ and $g(<x, y>) = <w, z>$.*

We will write $(A, R_A) \leq_{wi} (B, R_B)$ when there exists such a witness isomorphic reduction.

A particular well-behaved type of NP-search problem is the class of Constraint Satisfaction Problems:

Definition 6. *The Constraint Satisfaction Problem CSP(C) is specified by*

1. *A finite domain. Without loss of generality we will consider CSP with domain $D_k = \{0, \dots, k-1\}$, for some $k \geq 2$.*
2. *A finite set C of templates. A template is a finite subset $A \subseteq D_k^{(r)}$ for some $r \geq 1$, called the arity of template A.*

An instance Φ of CSP(C) is a formula obtained as a conjunction of instantiations of templates from C to tuples of variables from a fixed set $x_1, \dots, x_n$.

Definition 7. *[31] A generalized assignment for a boolean CNF formula is a mapping σ from the variables of F to $\{0, 1, *\}$. Given a generalized assignment σ for F, variable x is called a supported variable under σ if there exists a clause C of F such that x is the only variable of C that is satisfied by σ, and all the other literals are assigned FALSE.*

For CSP over general domains the previous definitions need to be generalized:

Definition 8. *(see also [49]) A generalized assignment for an instance of CSP(C) over domain $D_k = \{0,1,\dots,k-1\}$ is a mapping σ from the variables of Φ to $\mathscr{P}(D_k)\setminus\{\emptyset\}$. σ is simply called an assignment if $|\sigma(v)| = 1$ for all variables v of Φ. Given assignment σ_1 and generalized assignment σ_2, we say that σ_2 is compatible with σ_1 (in symbols $\sigma_1 \subseteq \sigma_2$) if for every variable v, $\sigma_1(v) \subseteq \sigma_2(v)$.*

For propositional formulas in CNF the important notions of *supported variable* and *cover* were introduced in [35], [31], [51]:

Definition 9. *Assignment $\sigma \in \{0,1,*\}^*$ is a cover of F iff:*

1. *every clause of F has at least one satisfying literal or at least two literals with value $*$ under σ ("no unit propagation"), and*
2. *σ has no unsupported variables assigned 0 or 1.*

It is called a true cover *(e.g. [31]) if there exists a satisfying assignment τ of F such that $\tau \subseteq \sigma$. In this paper we will only deal with true covers.*

One extension of the notion of supported variable to general CSP could be:

Definition 10. *Given a generalized assignment σ for an instance F of CSP(C), variable x is called supported by σ if there exists a constraint C of F such that*

1. *$|\sigma(v)| = 1$ for every $v \in C$.*
2. *for every $\lambda \neq \sigma(x)$, changing the value of $\sigma(x)$ to λ results in a generalized assignment that does not satisfy C.*

As for the extension of the notion of cover to general CSP, we have to modify Definition 9 for two reasons: first, we need more than three symbols to encode all the $2^k - 1$ possible choices in assigning a variable. Second, in the general case constraints cannot be satisfied by setting a single variable.

Definition 11. *A generalized assignment $\sigma \in D_k^n$ of a CSP F is a cover of F iff:*

- *no constraint of F can further eliminate any values of σ for its variables by consistency ("no unit propagation"; see also [50]).*
- *σ has no unsupported variables v (in the sense of Definition 10) s.t. $|\sigma(v)| = 1$.*

Definition 12. *Given NP-decision problems (A,R_A) and (B,R_B), a witness-isomorphic reduction of (A,R_A) to (B,R_B) is called* ***overlap preserving*** *if there is a continuous, monotonically increasing function $h:[0,1]\to[0,1]$ with $h(0)=0, h(1)=1$ such that for all x, $|x|=n$, and $z_1 \neq z_2$ such that $R_A(x,z_1)$ and $R_A(x,z_2)$,*

$$overlap(\pi_2^2 \circ g(x,z_1), \pi_2^2 \circ g(x,z_2)) = h(overlap(z_1,z_2)) + o_n(1).$$

In other words given two solutions z_1, z_2 of the instance x of A, one can map them (via the reduction) to two solutions w_1, w_2 of instance $f(x)$ of B such that the overlap of w_1, w_2 depends predictably on the overlap of the original instances z_1, z_2. In particular if x has a single cluster of solutions with a single overlap (respectively many clusters of solutions with a discontinuous overlap distribution) as predicted by the 1-RSB ansatz for random k-SAT) then the solution space of $f(x)$ should have a similar geometry.

Definition 13. *A wi-reduction (f,g) between search problems A and B is called* ***cover-preserving*** *if there exists a polynomial-time computable mapping $\overline{g}$ such that*

1. *$\overline{g}$ extends g, i.e. if x is an instance and y is a witness for x then $\overline{g}(<x,y>) = g(<x,y>)$.*
2. *$(f,\overline{g})$ is witness isomorphic when seen as a reduction* ***between covers.***
3. *$\overline{g}$ is compatible with g. That is, for all $x \in A$, for all y witnesses for x, for all generalized assignments z compatible with y, if z is a cover of x then $\overline{g}(<x,z>)$ is a cover of $f(x)$ compatible with $g(<x,y>)$.*

We will informally refer to reductions that have both properties as *geometry preserving*.

3 Geometry-preserving reductions via local constructions

Our first example of a geometry-preserving reduction is the classical reduction from k-coloring to k-SAT. The reduction is the usual one: to each vertex we associate three logical variables $x_{i,1}, x_{i,2}, x_{i,3}$. To each edge $e = (v_i, v_j)$ we associate a formula $F_e = F_e(x_{i,1}, x_{i,2}, x_{i,3}, x_{j,1}, x_{j,2}, x_{j,3})$. The formula F_e codifies three types of constraints: each vertex is given at least one color; no vertex is given more than one color; finally adjacent vertices cannot get the same color. That is

$$F_e = G(x_{i,1}, x_{i,2}, X_{i,3}) \wedge G(x_{j,1}, x_{j,2}, x_{j,3})) \wedge (\wedge_{k \neq l \in \{1,2,3\}} H(x_{i,1}, x_{i,2}) \wedge H(x_{j,k}, x_{j,l})) \qquad (2)$$

Theorem 1. *The natural reduction from k-COL to k-SAT is cover-preserving and overlap-preserving.*

Proof. Let G be an instance of k-COL and Φ_G the instance of k-SAT that is the image of G under the natural reduction. Let $c \in [k]^{|V(G)|}$ be a coloring of G and w be a generalized assignment (a.k.a. list coloring) compatible with c. Under the natural transformation each vertex v is represented by k logical variables, $x_{v,1}, x_{v,2}, \ldots, x_{v,k}$, and the constructed formula ensures that only one of them can be true. In particular one can define $f(c)$ to be the satisfying assignment naturally corresponding to the coloring c.

Now transform a general assignment w of colors to the graph G to a generalized assignment $g(G,w)$ for the formula by the following rules:

1. if $\emptyset \neq C_v \subseteq \{1,2,\ldots,k\}$ is a set of allowed colors for vertex v then we set all variables $x_{v,i}$, $i \notin C_v$, to zero.
2. if $|C_v| = 1$ then we set the remaining variable $x_{v,i}$ to 1. If, on the other hand, $|C_v| \geq 2$, then we set all variables $x_{v,i}$, $i \in C_v$ to $*$.

It is easy to see that w is a cover compatible with c if and only if $g(< G, w >)$ is a cover compatible with $g(< G, c >)$. □

Theorem 2. *The natural reduction from 1-in-k SAT to k-SAT, $k \geq 3$ is overlap preserving and cover preserving.*

Proof. Remember, clause 1-in-3(x,y,z) is translated as $x \vee y \vee z, \overline{x} \vee \overline{y} \vee \overline{z}$. Since the reduction does not add any new variables, the set of satisfying assignments is the same, hence the overlap of the translated solutions does not change. Similarly, covers correspond naturally (via identity) to covers. □

3.1 An(other) example of an overlap-preserving reduction

The purpose of this section is to give an example of a somewhat less trivial witness-isomorphic reduction that is overlap-preserving. The reduction is one that was already proved to be witness-isomorphic in the literature. Specifically, Fischer [17] has proved that problem MAX-SAT* (see Definition 4) is NP-complete. Subsequently Fischer, Hemaspaandra and Torenvliet [18] have shown that the reduction between SAT and $MAX - SAT^*$ is witness isomorphic.

Theorem 3. *One can encode $MAX - SAT^*$ as a NP-search problem such that the witness-isomorphic reduction from (SAT, R_{SAT}) to (MAX-SAT*,$R_{MAX-SAT^*}$) [18] is overlap-preserving.*

Proof. First let's recall the reduction from [18]: let ϕ be a propositional formula in CNF with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. Define

$$\psi = \bigwedge_{i=1}^{m} (x_i \vee b_i)^{m+1} \wedge (C_1 \vee \alpha) \wedge \ldots \wedge (C_m \vee \alpha)$$

In this formula $b_1,\dots,b_m,\alpha$ are new variables C^{m+1} denotes the fact that clause C has weight $(m+1)$, while $C\vee\alpha$ denotes the clause obtained by adding α to the disjunction in C. A clause without an upper index has weight 1. Let

$$\xi=\bigwedge_{j=1}^{n-1}\bigwedge_{i=j+1}^{n}(x_j\vee b_j\vee\overline{x_i})^{3m}\wedge(x_j\vee b_j\vee\overline{b_i})^{3m}.$$

Finally, let $\Psi=\psi\wedge\xi$.

It is easy to see (and was proved explicitly in [17]) that witnesses for instance $(\Psi,0^{2n+1},0^n)$ of MAX-SAT* correspond to satisfying assignments A for Φ as follows: we start at $\tau_0=0^{2n+1}$. First we flip either variable x_1 (if $A(x_1)=1$) or b_1 (if $A(x_1)=0$). Next we flip either variable x_2 (if $A(x_2)=1$) or b_2 (if $A(x_2)=0$). .... Finally, we flip either variable x_n (if $A(x_n)=1$) or b_n (if $A(x_n)=0$). Denote by P_A the path obtained in this way.

Our goal is to encode path P_A in such a way that for two assignments A,B, $d_H(P_A,P_B)=(2n+2)\cdot d_H(A,B)$. Since we encode one bit of a satisfying assignment by $(2n+2)$ bits of the encoding of paths, this relation proves that the reduction is overlap preserving. Indeed, $overlap(A,B)=\frac{d_H(A,B)}{n}$, and $overlap(P_A,P_B)=\frac{(2n+2)\cdot d_H(A,B)}{n(2n+2)}=overlap(A,B)$, so we can take $h(x)=x$ in Definition 12.

The idea of the encoding is simple: we use strings $z_1z_2\dots z_d$, where each z_i has length $2n+2$ and contains $n+1$ consecutive ones (including circular wrapping). The idea is that we encode into string z_i the event of flipping the value of the j'th variable in a fixed ordering of variables of Φ (specified below) by making z_i consist of $n+1$ consecutive ones, starting at position j. To make the encoding preserve the overlap predictably, make the position of variable b_j differ by $n+1$ (modulo $2n+2$) from the position of variable x_j.

Consider now two satisfying assignments A and B. If A and B agree on whether to flip x_i or b_i then the corresponding i'th blocks of P_A,P_B agree. Otherwise the corresponding blocks of P_A,P_B are complementary. So indeed we have $d_H(P_A,P_B)=(2n+2)\cdot d_H(A,B)$. □

3.2 An(other) example of a cover-preserving reduction

Whereas in the previous section we gave a slightly less trivial reduction that was overlap-preserving, the goal of this subsection is to give a slightly less trivial example of a cover-preserving reduction. The problem we are dealing with will not fall in the framework of Definition 6 (and subsequent Definitions 6–12) since it also includes a "global" constraint. Rather than attempting to rewrite many of the previous definitions in order to construct a more general framework, appropriate to our example, we will be content to keep things at an intuitive level, adapting the definitions on the spot as needed. We trace the idea of the construction below to an unpublished, unfinished manuscript [45] that has no realistic chances of ever being completed, as a (different) analysis of the belief propagation algorithm for graph bisection using methods from Statistical Physics has been published in the meantime [48]. The adaptation of the construction to our (rather different) purposes and the statement/proof of the theorem below is, however, entirely ours.

Definition 14. *The Perfect graph bisection (PGB) problem is specified as a search problem as follows:*

- **Input:** *A graph $G=(V,E)$.*
- **Witnesses:** *A function $f:V\to\{-1,1\}$ such that for all vertices $v,w\in V$, $(v,w)\in E\Rightarrow f(v)=f(w)$ and $\sum_{v\in V}f(v)=0$.*

We identify solutions f,g such that $f(v)=-g(v)$ for all $v\in V$.

Definition 15. *To define PGB as a local search problem we will work with partitions, defined as functions $f : V \to \{-1, 1\}$. Two partitions are adjacent if one can be obtained from the other by flipping two vertices on opposite sides of the partition. Covers are defined as strings over $\{-1, 1, *\}$, where again we identify strings that only differ by permuting labels -1 and 1.*

The first step in order to set up PGB as a local search problem subject to cover-preserving reductions is to represent G as a *factor graph* [33]. A factor graph is a bipartitite undirected graph containing two different kinds of nodes: *variable nodes*, corresponding to vertices v_i in G, and *function nodes* corresponding to edges e_{ij} in G. A variable node and function node are connected in the factor graph if the edge represented by the function node is incident on the vertex represented by the variable node. It is clear that a function node is adjacent to exactly two variable nodes.

The constraint on edges only connecting vertices in the same subset is enforced in an entirely straightforward and local manner in the factor graph: a function node requires both its neighboring variable nodes to be assigned the same value $s = +1$ or -1. The balance constraint $\sum_{i=1}^{n} s_i = 0$, however, remains a global one.

Therefore, the next step is to give a representation of perfect graph bipartition that allows balance to be enforced in a purely local manner.

Definition 16. *Given r variables that can take values $\{-1, 0, 1\}$, the* counting constraint $CC_m(x_1, \ldots, x_r)$ *is true if and only if $\sum_i x_i \in \{\pm m\}$.*

Definition 17. *An instance Ψ of a counting CSP problem consists of n variables connected by a set of counting constraints CC_{n-4}. The variables can take values $\{-1, 0, 1\}$. A witness for the counting CSP Ψ is an assignment of values to variables such that all constraints are satisfied.*

Theorem 4. *There exists a cover-preserving, overlap-preserving reduction from PGB to counting CSP.*

Proof. Consider an instance of G of GBP as described above. We translate it into an instance Ψ_G of counting CSP by constructing a new factor graph with $\binom{n}{2}$ variable nodes v_{kl}, $1 \le k < l \le n$. Whereas a function node e_{ij} was previously connected to 2 variable nodes v_i and v_j, it is now connected to $2n-4$ variable nodes. Taking the case where $2 < i < j-1 < n-2$, they are: $v_{1,i}, v_{2,i}, \ldots, v_{i-1,i}, v_{i,i+1}, \ldots, v_{i,j-1}$, $v_{i,j+1}, \ldots, v_{i,n-1}, v_{in}$ and $v_{1j}, v_{2j}, \ldots, v_{i-1,j}, v_{i+1,j}, \ldots, v_{j-1,j}$, $v_{j,j+1}, \ldots$, $v_{j,n-1}, v_{j,n}$. Other cases ($j > i$, etc.) may be treated similarly: only the labeling details change. Note that variable node v_{ij} is *not* connected to function node e_{ij}. The reasons for this will become apparent shortly.

In our new factor graph, assign to each of the $\binom{n}{2}$ variable nodes v_{kl} a value $x_{kl} = (s_k + s_l)/2$. Note that $x_{kl} \in \{-1, 0, +1\}$, depending on whether $s_k = s_l = -1$, $s_k \neq s_l$, or $s_k = s_l = +1$. Next, we will show how the graph bisection constraints translate into a local constraint on the values of x_{kl} for all variable nodes connected to a given function node e_{ij}.

We start by providing local conditions in the new factor graph that are necessary for the perfect graph bisection constraints to be satisfied. We then show that these conditions are not only necessary but also sufficient.

Consider the sum of the values x_{kl} for all $2n-4$ variable nodes connected to a function node e_{ij}:

$$\sum_{\substack{k,l:\} \\ v_{kl} \sim e_{ij}}} x_{kl} \;=\; \sum_{m=1}^{i-1} x_{mi} + \sum_{\substack{m=i+1 \\ m \neq j}}^{n} x_{im} + \sum_{\substack{m=1 \\ m \neq i}}^{j-1} x_{mj} + \sum_{m=j+1}^{n} x_{jm} = \sum_{\substack{m=1 \\ m \neq i,j}}^{n} \frac{(2s_m + s_i + s_j)}{2} = \sum_{m=1}^{n} s_m + \frac{(n-4)(s_i + s_j)}{2}.$$

For a balanced solution, $\sum_{m=1}^{n} s_m = 0$. Furthermore, since v_i and v_j are connected by an edge e_{ij}, they must be assigned to the same partition, and so $s_i = s_j$. Thus, the necessary condition for the graph bisection constraints to be satisfied is that for all function nodes e_{ij},

$$\sum_{k,l:v_{kl}\sim e_{ij}} x_{kl} = \pm(n-4). \tag{3}$$

The construction of Ψ_G is now clear: it is the conjunction of counting constraints specified by equation (3). We identify two solutions of Ψ_G that can be obtained by multiplying all values by -1.

First we have to show that what we constructed is indeed a reduction. That is, we first show that except at small n and for a very small number of graphs, the conjunctions of conditions (3) is also a sufficient condition for the existence of a perfect bipartition in G. Suppose, indeed that (3) are satisfied but that the graph bisection constraints are not. There are three ways this can happen: 1) there is a balanced solution with an edge constraint violated, 2) there is an unbalanced solution with no edge constraints violated, and 3) there is an unbalanced solution with an edge constraint violated.

Consider case 1. If there is a balanced solution with an edge e_{ij} such that $s_i \neq s_j$, $\sum_{k,l:v_{kl}\sim e_{ij}} x_{kl} = 0$. This is clearly inconsistent with (3) unless $n = 4$.

Now consider case 2. The only way that (3) can be respected when there is an imbalance and no edge constraints violated is if $\sum_{m=1}^{n} s_m = \pm 2(n-4)$. Since $-n \leq \sum_{m=1}^{n} s_m \leq +n$, this is impossible unless $n \leq 8$.

Finally, consider case 3. In this case, (3) could still be respected if $\sum_{m=1}^{n} s_m = \pm(n-4)$. For that to happen, however, 2 vertices must be assigned to one partition and $n-2$ vertices to the other, and the *all* edges in the graph must be violated. In that case, no edges can be contained within either partition.

Note that solutions $f : V \to \{-1,1\}$ of G naturally correspond to solutions $x^{(f)}$, $x^{(f)}_{k,l} = \frac{f(k)+f(l)}{2}$, of the corresponding formula Ψ_G. Now the fact that the reduction is cover-preserving is easy: a cover c for a solution f consists of specifying for some pairs of vertices of G whether they are always on the same side of a perfect bipartition, whether they are always on opposite sides of a perfect bipartition, or whether they can fluctuate between sides. We construct a cover $\tilde{c}$ for $x^{(f)}$ corresponding to c as follows:

- if two vertices k,l are always on the same side of the partition in any perfect bipartition then the variable $x^{(f)}_{k,l}$ never takes value 0. In this case $\tilde{c}_{k,l} = \mathbf{1}$.
- If they always are on opposite sides then the variable $x_{k,l}$ always takes value 0. So we set $\tilde{c}_{k,l} = \mathbf{0}$.
- Otherwise $x^{(f)}_{k,l}$ may take all three values. We set $\tilde{c}_{k,l} = *$.

As for overlaps, given two perfect bipartitions f,g of G, we define

$$overlap(f,g) = \frac{1}{n} min(d_H(f,g), d_H(-f,g)). \tag{4}$$

Similarly, given two solutions x,y for Ψ_G define

$$overlap(x,y) = \frac{1}{\binom{n}{2}} min(d_H(x,y), d_H(-x,y)). \tag{5}$$

In equations (4) and (5) we take into account, of course, the factoring of the solution spaces by equivalence. The proportionality factor of (5) is $\frac{1}{\binom{n}{2}}$ since formula Ψ_G has that many variables. Now it is easy to see that

$$d_H(x^{(f)}, x^{(g)}) = \frac{\binom{n \cdot d_H(f,g)}{2}}{\binom{n}{2}} = (d_H(f,g))^2 + o_n(1) \tag{6}$$

and similarly for $-f$ and g, so we can take $h(x) = x^2$. Indeed, if f can be made to coincide on v vertices with g then $x^{(f)}$ and $x^{(g)}$ coincide on all pairs $x^{(f)}_{k,l}$ and $x^{(g)}_{k,l}$, where k,l range over the set of variables where k,l coincide. If f,g differ on one of k,l or both then $x^{(f)}_{k,l} \neq x^{(g)}_{k,l}$. So relation (6) is clear. □

4 Reductions that are not geometry-preserving

A basic sanity check is to verify that there are natural reductions between versions of satisfiability that are not geometry preserving. In this section we give two examples:

Theorem 5. *The classical reduction from 4-SAT to 3-SAT is neither cover preserving nor overlap preserving.*

Proof. Remember, the idea of simulating a 4-clause $x \vee y \vee z \vee t$ by a 3-CNF formula is to add an extra variable α and simulate the clause above by the formula $x \vee y \vee \alpha, \overline{\alpha} \vee z \vee t$.

To show that the reduction is not cover preserving it is enough to create a formula such that by the reduction we create more covers.

We accomplish this as follows: we add to the formula clause $C_1 = x \vee y \vee z \vee t$. Then, informally, we enforce constraints $x \vee y$ and $z \vee t$ using 4-clauses. Let us accomplish this for the first one (the second case is entirely analogous) as follows: we add clauses $x \vee y \vee x_1 \vee x_2, x \vee y \vee \overline{x_1} \vee x_2, x \vee y \vee x_1 \vee \overline{x_2}, x \vee y \vee \overline{x_1} \vee \overline{x_2}$. Let Φ_1 be the resulting formula and Φ_2 be its 3-CNF reduction.

The set of solutions of Φ_1 is basically $\{(x,y) \in \{(0,1),(1,0),(1,1)\}\} \times \{(z,t) \in \{(0,1),(1,0),(1,1)\}\}$. None of the variables x,y,z,t can be given a 0/1 value in a cover since they are not supported in clause C_1 (as there are at least two true literals in that clause). So the set of covers of Φ_1 coincides with the set of satisfying assignments of Φ, together with the trivial cover $*** \ldots **$.

On the other hand, in Φ_2 auxiliary variable α can be given value $*$. In particular $x = 1, z = 1, \alpha = *$ and all other variables given value $*$ is a nontrivial cover. As all values for α are good, the reduction is also not witness isomorphic, hence not overlap preserving. □

5 Geometry-preserving reductions: an Algebraic Perspective

In this section we advocate for the development of an algebraic approach to reductions, perhaps based on *category theory* [34]. The motivations for advocating such an approach are simple:

- One can view each constraint satisfaction problem $CSP(S)$ as a category: its objects are S-formulas. Given two objects (formulas) Φ_1 and Φ_2, a morphism is a mapping between variables of Φ_1 and variables of Φ_2 that maps clauses of Φ_1 to clauses of Φ_2.
- CSP's, together with polynomial time reductions form a category $\mathscr{CSP}_{\leq_m^P}$.
- We can view our constructions in Section 3, however simple, in a categorical fashion. The proof of Theorem 1 shows that the reduction "localizes": we can interpret the usual gadget-based reduction (and in particular our cover reduction) as "gluing" local morphisms. Then it is possible to construct cover-preserving reductions by:
  1. Adequately constructing them "locally".
  2. "Gluing" them in a way that preserves covers.

The following result shows that cover-preserving reductions also can act as morphisms:

Theorem 6. *The following (simple) properties are true:*

1. *For any CSP A the identity map is a cover-preserving reduction of A to itself.*
2. *Cover-preserving reductions compose: if A,B,C are CSP and $\leq_{A,B}$, $\leq_{B,C}$ are cover-preserving reductions of A to B and B to C, respectively, then $\leq_{A,C} := \leq_{B,C} \circ \leq_{A,C}$ is a cover-preserving reduction of A to C.*

Consequently, constraint satisfaction problems with cover-preserving reductions as morphisms form a category $\mathscr{CSP}_{cover}$, indeed a subcategory of $\mathscr{CSP}_{\leq_m^P}$.

Similar results hold for overlap-preserving reductions.

Proof. 1. This is trivial.

2. Let x be an instance of A, y be the instance of B that corresponds to x via $\leq_{A,B}$ and z be the instance of C that corresponds to y via $\leq_{B,C}$. Further, let a, a_1 be strings. a is a cover of a witness a_1 for x, if and only if $b := \leq_{A,B}(a)$ is a cover of witness $b_1 = \leq_{A,B}(a_1)$ of instance $y = \leq_{A,B}(x)$. This last statement happens if and only if $c := \leq_{B,C}(b)$ is a cover of witness $c_1 := \leq_{B,C}(b_1)$ of instance $z = \leq_{B,C}(y)$.

   But $z = \leq_{B,C} \circ \leq_{A,B}(x)$, $c_1 = \leq_{B,C} \circ \leq_{A,B}(a_1)$ and $c = \leq_{B,C} \circ \leq_{A,B}(a)$, so the composition of reductions is a reduction and cover preserving. It is clearly specified by a polynomial computable function.

Similar arguments function for overlap-preserving reductions. Here we use the fact that the composition of monotonically increasing, continuous functions from [0,1] to [0,1] (such that h(0)=0, h(1)=1) has the same properties. □

Note that there have recently been several attempts to consider ordinary reductions in computational complexity theory from a structural perspective [36, 43, 37]. What can be done for ordinary reductions might be interesting, we believe, for (variants of) our reductions as well.

6 Conclusions

Our research clearly has a preliminary character: we see our contribution as a conceptual one, pointing in the right direction rather than having provided the definitive definitions for structure-preserving reductions. Specifically, we believe that one needs to be serious about connecting phase transitions in Combinatorial Optimization to Computational Complexity Theory, however weak the resulting notions may turn out to be. ([16] seems to be an intriguing approach in this direction).

It may be that the results in this paper can be redone using a more restrictive notion of reductions that turns out to be better suited. Whether different variations on the concepts are more appropriate is left for future research. Also left for further research is the issue of classifying the natural encodings of constraint satisfaction problems (particularly for NP-complete problems) under these definitions. This might be difficult, though: our result only shows, for instance, that the natural reduction from 4-SAT to 3-SAT does not have the desired properties, *not that there is no such reduction !*. Also, one cannot solve this problem, in principle without understanding the nature of the phase transition in 3-SAT: the 1-RSB approach predicts the phase transition of k-SAT only for large enough values of k, and it is not clear that it does for $k = 3$.

In any case we expect that NP-complete problems split in multiple degrees under (variants of) our reductions. Reasons for this belief are twofold:

- First, it is actually known that not all NP-complete problems are witness isomorphic. So we cannot expect to have a single degree under cover-preserving reductions.
- Second of all, we expect that 1-in-k SAT and k-SAT are **not** equivalent under overlap preserving reductions, simply because k-SAT displays clustering below the phase transition, whereas we expect 1-in-k SAT, $k \geq 3$, not to. Whether our notions of reductions are appropriate enough to capture this difference is an open problem.

The main weakness of any attempt of connecting the geometry of instances of two problems by reductions is the asymptotic nature of the predictions provided by methods from Statistical Physics. Specifically, such predictions (the existence of a sharp SAT/UNSAT threshold, the existence of one/multiple clusters of solutions, etc.) refer to properties that are true for **almost all** (rather than all) instances at a given density. In contrast classical reductions (as well as the ones we give) provide guarantees for **all** instances.

Secondly, our definitions have not attempted to map (via the reduction) the precise location of the two phase transitions, and there is no reasonable expectation that ordinary reductions preserve this location. A reason is that, for instance, $(2+p)$-SAT [41] is NP-complete for $p>0$, but the location of the phase transition in $(2+p)$-SAT is determined [41, 2] by the "easy part", at least for $p<0.4$. So, unless the reduction classifies such problems as "easy" ("reducing" an instance Φ of $(2+p)$-SAT to its 2-SAT component via a notion of reduction that may sometimes err, but is otherwise rather trivial) one cannot simply use ordinary notions of reductions, even restricted as the ones put forward in this paper[3].

To conclude: to get the notion of reduction appropriate for dealing with phase transitions we might need a notion of reduction that is only correct on one class of instances (and errs on a negligible set of instances at each density). We don't claim that our reductions accomplish this, and only hope that they can function as useful guidelines towards the development of this "correct" notions of reductions. How to do this (and whether it can be done at all) is intriguing. And open.

References

[1] D. Achlioptas, A. Chtcherba, G. Istrate & C. Moore (2001): *The phase transition in 1-in-K SAT and NAE3SAT*. In: *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*, pp. 719–722, doi:10.5555/365411.365760.

[2] D. Achlioptas, L.M. Kirousis, E. Kranakis & D. Krizanc (2001): *Rigorous results for random $(2+p)$-SAT*. *Theoretical Computer Science* 265(1–2), pp. 109–129, doi:10.1016/S0304-3975(01)00154-2.

[3] D. Achlioptas & F. Ricci-Tersenghi (2006): *On the solution space geometry of random constraint satisfaction problems*. In: *Proceedings of the 36th ACM Symposium on Theory of Computing*, pp. 130–139, doi:10.1145/1132516.1132537.

[4] Dimitris Achlioptas & Michael Molloy (2015): *The solution space geometry of random linear equations*. *Random Structures & Algorithms* 46(2), pp. 197–231, doi:10.1002/rsa.20494.

[5] Manindra Agrawal, Eric Allender, Russell Impagliazzo, Toniann Pitassi & Steven Rudich (2001): *Reducing the complexity of reductions*. *Computational Complexity* 10(2), pp. 117–138, doi:10.1007/s00037-001-8191-1.

[6] L. Berman & J. Hartmanis (1977): *On Isomorphisms and Density of NP and Other Complete Sets*. *SIAM Journal on Computing* 6(2), pp. 305–322, doi:10.1137/0206023.

[7] A. Braunstein, M. Mézard & R. Zecchina (2005): *Survey propagation: an algorithm for satisfiability*. *Random Structures and Algorithms* 27(2), pp. 201–226, doi:10.1002/rsa.20057.

[8] Guy Bresler & Brice Huang (2021): *The algorithmic phase transition of random k-SAT for low degree polynomials*. In: *62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS'21)*, pp. 298–309, doi:10.1109/FOCS52979.2021.00038.

[9] Andrei A Bulatov (2017): *A dichotomy theorem for nonuniform CSPs*. In: *58th Annual IEEE Symposium on Foundations of Computer Science (FOCS'17)*, pp. 319–330, doi:10.1109/FOCS.2017.37.

[3]In the journal version of the paper, to be posted on arxiv in the near future, we plan, in fact, to develop such "reductions"

[10] Amin Coja-Oghlan, Tobias Kapetanopoulos & Noela Müller (2020): *The replica symmetric phase of random constraint satisfaction problems*. Combinatorics, Probability and Computing 29(3), pp. 346–422, doi:10.1017/S0963548319000440.

[11] N. Creignou & H. Daudé (2004): *Coarse and Sharp Thresholds for Random Generalized Satisfiability Problems*. In M. Drmota et al., editor: Mathematics and Computer Science III: Algorithms, Trees, Combinatorics and Probabilities, Birkhauser, pp. 507–517, doi:10.1007/978-3-0348-7915-6.

[12] Nadia Creignou (1995): *The class of problems that are linearly equivalent to satisfiability or a uniform method for proving NP-completeness*. Theoretical Computer Science 145(1-2), pp. 111–145, doi:10.1016/0304-3975(94)0018.

[13] Víctor Dalmau & Jakub Opršal (2024): *Local consistency as a reduction between constraint satisfaction problems*. In: Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '24, doi:10.1145/3661814.3662068.

[14] H. Daudé, M. Mézard, T. Mora & R. Zecchina (2008): *Pairs of SAT assignments in random boolean formulae*. Theoretical Computer Sci. 393(1-3), pp. 260–279, doi:10.1016/j.tcs.2008.01.005.

[15] Jian Ding, Allan Sly & Nike Sun (2022): *Proof of the satisfiability conjecture for large k*. Annals of Mathematics 196(1), pp. 1–388, doi:10.4007/annals.2022.196.1.1.

[16] Alexander Durgin (2020): *CSP-Completeness And Its Applications*. M.Sc. Thesis, Washington University in Saint Louis, doi:10.7936/2sbr-dv81.

[17] S. Fischer (1995): *A note on the Complexity of Local Search Problems*. Information Processing Letters 53(1), pp. 69–75, doi:10.1016/0020-0190(94)00184-Z.

[18] S. Fischer, L. Hemaspaandra & L. Torenvliet (1997): *Witness-Isomorphic Reductions and Local Search*. In M. Sorbi, editor: Complexity, Logic and Recursion Theory, Lecture Notes in Pure and Applied Mathematics, Marcel Dekker, pp. 207–223, doi:10.1201/9780429187490.

[19] David Gamarnik (2021): *The overlap gap property: A topological barrier to optimizing over random structures*. Proceedings of the National Academy of Sciences 118(41), p. e2108492118, doi:10.1073/pnas.2108492118.

[20] David Gamarnik & Eren C Kızıldağ (2023): *Algorithmic obstructions in the random number partitioning problem*. The Annals of Applied Probability 33(6B), pp. 5497–5563, doi:10.1214/23-AAP1953.

[21] Hamed Hatami & Michael Molloy (2008): *Sharp thresholds for constraint satisfaction problems and homomorphisms*. Random Structures & Algorithms 33(3), pp. 310–332, doi:10.1002/rsa.20225.

[22] Harry B Hunt III, Madhav V Marathe & Richard E Stearns (2001): *Strongly-local reductions and the complexity/efficient approximability of algebra and optimization on abstract algebraic structures*. In: Proceedings of the 46th International Symposium on Symbolic and Algebraic Computation (ISSAC'21), pp. 183–191, doi:10.1145/384101.384126.

[23] H.B. Hunt III, R. Stearns & M.V. Marathe (2000): *Relational Representability, Local Reductions and the Complexity of Generalized Satisfiability Problems*. Technical Report LA-UR-006108, Los Alamos National Laboratory.

[24] G. Istrate (2000): *Computational Complexity and Phase Transitions*. In: Proceedings of the 15th I.E.E.E. Annual Conference on Computational Complexity (CCC'00), pp. 104–115, doi:10.1109/CCC.2000.856740.

[25] G. Istrate (2002): *The phase transition in random Horn satisfiability and its algorithmic implications*. Random Structures and Algorithms 4, pp. 483–506, doi:10.1002/rsa.10028.

[26] G. Istrate (2004): *On the Satisfiability of random k-Horn formulae*. In P. Winkler & J. Nesetril, editors: Graphs, Morphisms and Statistical Physics, pp. 113–136, doi:10.1090/dimacs/063.

[27] G. Istrate (2005): *Threshold properties of random boolean constraint satisfaction problems*. Discrete Applied Mathematics 153, pp. 141–152, doi:10.1016/j.dam.2005.05.010.

[28] G. Istrate (2007): *Satisfiability of random Boolean CSP: Clusters and Overlaps*. Journal of Universal Computer Science 13(11), pp. 1655–1670, doi:10.3217/jucs-013-11-1655.

[29] G. Istrate, A. Percus & S. Boettcher (2005): *Spines of Random Constraint Satisfaction Problems: Definition and Connection with Computational Complexity*. Annals of Mathematics and Artificial Intelligence 44(4), pp. 353–372, doi:10.1007/s10472-005-7033-2.

[30] Gabriel Istrate (2009): *Geometric properties of satisfying assignments of random ε-1-in-k SAT*. International Journal of Computer Mathematics 86(12), pp. 2029–2039, doi:10.1080/00207160903193970.

[31] L. Kroc, A. Sabharwal & B. Selman (2007): *Survey Propagation Revisited*. In: Proceedings of the 23rd Conf. on Uncertainty in Artificial Intelligence (UAI'07), pp. 217–226, doi:10.5555/3020488.3020515.

[32] Florent Krzakała, Andrea Montanari, Federico Ricci-Tersenghi, Guilhem Semerjian & Lenka Zdeborová (2007): *Gibbs states and the set of solutions of random constraint satisfaction problems*. Proceedings of the National Academy of Sciences 104(25), pp. 10318–10323, doi:10.1073/pnas.0703685104.

[33] F.R. Kschischang, B.J. Frey & H.-A. Loeliger (2001): *Factor graphs and the sum-product algorithm*. IEEE Trans. Infor. Theory 47, pp. 498–519, doi:10.1109/18.910572.

[34] S. Mac Lane (1998): *Categories for the Working Mathematician (second edition)*. Springer Verlag, doi:10.1073/pnas.0703685104.

[35] Elitza Maneva, Elchanan Mossel & Martin J. Wainwright (2007): *A new look at survey propagation and its generalizations*. Journal of the A.C.M. 54(4), p. 17, doi:10.1145/1255443.1255445.

[36] Damiano Mazza (2018): *A Functorial Approach to Reductions among Decision Problems*. Abstract presented at DICE'2018.

[37] Damiano Mazza (2022): *A Categorical Approach to Descriptive Complexity Theory*. Abstract presented at the Structure Meets Power Workshop, p. 9.

[38] M. Mézard & A. Montanari (2007): *Information, Physics and Computation*. Oxford University Press, doi:10.1093/acprof:oso/9780198570837.001.0001.

[39] M. Mézard, T. Mora & R. Zecchina (2005): *Clustering of solutions in the random satisfiability problem*. Physical Review Letters 94(197205), doi:10.1103/PhysRevLett.94.197205.

[40] M. Molloy (2002): *Models for random constraint satisfaction problems*. In: Proceedings of the 32nd ACM Symposium on Theory of Computing, doi:10.1137/S0097539700368667.

[41] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman & L. Troyansky (1999): *Determining computational complexity from characteristic phase transitions*. Nature 400(8), pp. 133–137, doi:10.1038/22055.

[42] C. Moore, G. Istrate, D. Demopoulos & M. Vardi (2007): *A continuous-discontinuous second-order transition in the satisfiability of a class of Horn formulas*. Random Structures and Algorithms 31(2), pp. 173–185, doi:10.1002/rsa.20176.

[43] Adam O Conghaile, Samson Abramsky & Anuj Dawar (2022): *A sheaf-theoretic approach to (P)CSPs*. Abstract presented at the Structure Meets Power Workshop, p. 3.

[44] A. Percus, G. Istrate & C. Moore, editors (2006): *Computational Complexity and Statistical Physics*. Oxford University Press, doi:10.1093/oso/9780195177374.001.0001.

[45] Allon Percus, Gabriel Istrate, Shiva Prasad Kasiviswanathan, Stefan Boettcher, Nicholas Hengartner & Bruno Gonçalves Tavares (2007): *Belief Propagation for Graph Bisection*. (unpublished manuscript).

[46] T. J. Schaefer (1978): *The complexity of satisfiability problems*. In: Proceedings of the 13th ACM Symposium on Theory of Computing (STOC'78), pp. 216–226, doi:10.1145/800133.804350.

[47] Iain A Stewart (1994): *On completeness for NP via projection translations*. Mathematical Systems Theory 27(2), pp. 125–157, doi:10.1007/BF01195200.

[48] Petr Šulc & Lenka Zdeborová (2010): *Belief propagation for graph partitioning*. Journal of Physics A: Mathematical and Theoretical 43(28), p. 285003, doi:10.1088/1751-8113/43/28/285003.

[49] Ronghui Tu, Yongyi Mao & Jiying Zhao (2008): *Survey Propagation as "Probabilistic Token Passing"*. IEICE Transactions on Information and Systems 91(2), pp. 231–233, doi:10.1093/ietisy/e91-d.2.231.

[50] Ronghui Tu, Yongyi Mao & Jiying Zhao (2010): *Is SP BP?* IEEE Transactions on Information Theory 56(6), pp. 2999–3032, doi:10.1109/ITW.2007.4313080.

[51] D. Vilenchik (2007): *It's all about the Support: A new perspective on the Satisfiability Problem*. Journal of Satisfiability, Boolean Modeling and Computation 3, pp. 125–139, doi:10.3233/SAT190033.

[52] Mihalis Yannakakis (1997): *Computational complexity*. In Local search in combinatorial optimization, pp. 19–55, doi:10.2307/j.ctv346t9c.7.

[53] Dmitriy Zhuk (2020): *A proof of the CSP dichotomy conjecture*. Journal of the A.C.M. 67(5), pp. 1–78, doi:10.1145/3402029.

Converting BPMN Diagrams to Privacy Calculus

Georgios V. Pitsiladis Petros S. Stefaneas

Department of Mathematics
School of Applied Mathematical and Physical Sciences
National Technical University of Athens
9 Iroon Polytechniou Str., 15772 Zografou, Greece

gpitsiladis@mail.ntua.gr petros@math.ntua.gr

The ecosystem of Privacy Calculus is a formal framework for privacy comprising (a) the Privacy Calculus, a Turing-complete language of message-exchanging processes based on the π-calculus, (b) a privacy policy language, and (c) a type checker that checks adherence of Privacy Calculus terms to privacy policies. BPMN is a standard for the graphical description of business processes which aims to be understandable by all business users, from those with no technical background to those implementing software. This paper presents how (a subset of) BPMN diagrams can be converted to Privacy Calculus terms, in the hope that it will serve as a small piece of larger workflows for building privacy-preserving software. The conversion is described mathematically in the paper, but has also been implemented as a software tool.

1 Introduction

The main motivation of this paper is that it might serve as a first version of a piece of a larger workflow for building privacy-preserving software.

In order to trust that some piece of software is privacy-preserving, this must somehow be proved formally; in other words, privacy protection needs to be considered as a formal specification (formalised privacy policies) complemented by tools able to decide adherence of programs to policies.

The Privacy Calculus ecosystem has been introduced in [10] to tackle these considerations; it was further developed in [11, 9, 17, 16, 12, 23]. Privacy Calculus is a variation of the π-calculus, a Turing-complete language describing parallel processes sharing messages. It is accompanied by a privacy policy language, which gives the ability to grant *permissions* (read, write, disclose, store, etc.) to *users* or *groups* (forming a hierarchy) for specific *purposes*[1]. The ecosystem is completed by a type checker for checking compliance of Privacy Calculus terms to privacy policies written in the aforementioned formal language.

Although some tools for working with the Privacy Calculus ecosystem have been created [18, 22], the ecosystem is still quite abstract, far from everyday practice. One way to bridge this gap is to create conversions between higher-level frameworks and Privacy Calculus. This is where BPMN might fruitfully enter the discussion.

The aim of Business Process Model and Notation (BPMN) is to serve as a standard for the graphical depiction of business processes, enhancing intra- and inter-organisational communication and interoperability. It is high-level enough to be understandable by audiences with minimal technical background, however it can be quite detailed and (in its full generality) even automatically executable.

This paper is an exploration of how the most basic elements of BPMN can be converted to Privacy Calculus terms with the hope that eventually, a workflow such as the following could be feasible: (1) describe a business process in BPMN, (2) convert it to Privacy Calculus, (3) specify a privacy policy, ideally

[1]The notion of purpose is inherent in privacy protection. This has been argued in the literature regarding privacy, but has also been acknowledged in practice by legislation: purpose of data processing is a fundamental notion in GDPR.

by converting it from some high-level framework to the formal privacy policy language, (4) obtain (e.g. with the Maude tool presented in [18]) a proof that the business process adheres to the policy.

The rest of this paper is organised as follows: Section 2 reviews basic notions of BPMN, Section 3 contains some basic definitions of the Privacy Calculus, and Section 4 discusses how BPMN diagrams (or rather, a subset of them) can be converted to Privacy Calculus terms and presents a tool that automates the said conversion; Section 5 contains some concluding remarks.

2 Business Process Model and Notation

BPMN defines, both syntactically and semantically, a multitude of graphical elements. These elements can be combined into diagrams. Three kinds of diagrams are possible: Collaborations, Processes, and Choreographies ([14, Section 1.1]); here, only the first two will be considered.

Processes can be public or private. Private Processes model activities within an organisation: they can be defined at a so high level of detail as to be executable; otherwise, they serve for documentation purposes. Public Processes are non-executable and show activities of multiple Participants, documenting their interaction and hiding (parts of) actions internal to Participants [14, Section 7.2.1]. Here, since the interest lies on data protection among multiple stakeholders (the data subject and at least one data processing entity), mostly public Processes will be considered. Figs. 1a and 1b are examples of Processes.

A Collaboration contains two or more Participants and its purpose is to depict the interactions among them [14, Section 7.2.1]. Each Participant is depicted as a Pool which may be empty or contain a Process diagram [14, Table 7.1] (at most one process can be private, in which case it may be drawn outside of a Pool [14, Section 9.3]). Pools can also be divided in Lanes and/or have multiple instances, but these features will not be considered here. Fig. 2 is an example of a Collaboration with two Pools.

There are five categories of graphical elements [14, Section 7.3]: flow objects (Events, Activities, Gateways), data (Data Objects, Data Stores), connecting objects (Sequence Flows, Message Flows, Associations, Data Associations), swimlanes (Pools, Lanes), and artifacts (Text Annotations, element Groups). Here, only flow objects, Flows, and Pools will be considered; the main characteristics of Events, Activities, Gateways, and Flows will be presented in Sections 2.1 to 2.4. Data Objects and Data Stores will not be considered, since the version of Privacy Calculus employed here cannot deal with them properly. Messages will only be considered indirectly, because they are not supported by the bpmn.io diagram editor; when needed, they will be considered as available externally to the BPMN modelling.

In order to understand how control flows within a diagram, the concept of *tokens* is employed in lieu of semantics; in the words of [14, Section 7.2], "A token is a *theoretical* concept that is used as an aid to define the behavior of a Process that is being performed. The behavior of Process elements can be defined by describing how they interact with a token as it 'traverses' the structure of the Process.". In short (and with many details omitted), Start Events generate tokens, End Events consume them, and the other elements redirect, multiply, or merge them appropriately. Tokens will be instrumental for the conversion to Privacy Calculus.

2.1 Events

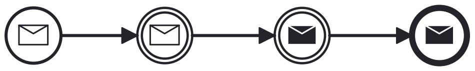
There are three types of Events[2] based on *when* they affect the flow of a process: Start, Intermediate, and End. There are also multiple types of events depending on *how* the affect the flow: here, only Message Events (and Start/End events with no information as to how they affect the flow) will be considered.

[2]BPMN also defines Events at the boundaries of Activities [14, Section 10.5.4], but these will not be considered here.

Every Event either catches or throws (but not both): Start Events always catch, End Events always throw, Intermediate Message Events may do either. [14, Section 7.3.2]

Contrary to [14, Section 10.5.2 and Section 7.2.1], which allow leaving Start/End events implicit for simplicity, this paper requires that Processes (and Sub-Processes) must always start with one or more Start Events and that each path of a Process (or Sub-Process) must terminate at an End Event. This affects expressiveness only minimally; moreover, in future treatments, "phantom" Start/End Events, connected to the "initial" and "final" Flow Nodes, could be added if none are provided.

Naturally, Start Events have no incoming Sequence Flows and End Events have no outgoing Sequence Flows. In order to accommodate implicit Start/End Events, BPMN ([14, Section 10.5.2]) allows this for other Flow Nodes as well. Here, the only Flow Nodes that will be permitted not to have an incoming Sequence Flow are Start Events; dually, the only Flow Nodes that will be permitted not to have an outgoing Sequence Flow are End Events and Sub-processes.


(a) A Process with only Message Events as Flow Nodes. From start to end, it contains a Message Start Event, a Message Intermediate Catch Event, a Message Intermediate Throw Event, and a Message End Event.

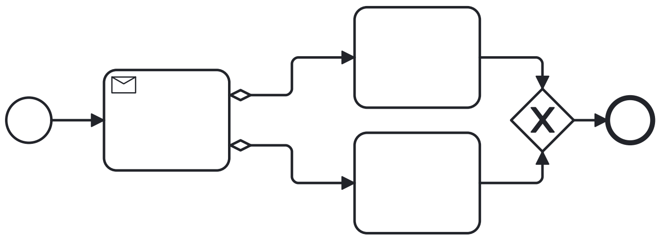

(b) A Process with some Conditional Flows (recognised by the diamond symbol at their start). The Start Event is followed by a Receive Task. Depending on the conditions, one or both of the following two Abstract Tasks are triggered; here, it is assumed that the conditions are such that only one can be fulfilled. An Exclusive Gateway combines the two alternative paths and channels the flow to the End Event.

Figure 1: Two diagrams of BPMN Processes.

2.2 Activities

Activities are divided into Tasks and Sub-Processes. Tasks are atomic (as far as the modelling is concerned), while Sub-Processes are compound [14, Section 7.3.2].

Tasks: A Task is an Activity which represents some action not broken down to more detail, hence considered atomic (in fact, it might be cancelled in mid-execution through the Compensation or other mechanisms of BPMN, but this is not considered here). There are many types of Tasks, including a generic one (Abstract Tasks). Apart from Abstract Tasks, this paper is mainly interested in Send Tasks (e.g. "Send confirmation receipt" in Fig. 2), which send Messages to other Participants, and Receive Tasks (e.g. "Listen for confirmation" in Fig. 2), which receive Messages from other Participants. Among the rest types of Tasks are User Tasks (e.g. "Receive notification" in Fig. 2), which are executed by humans with the aid of automated systems, and Manual Tasks, which are executed by humans manually (e.g. "Send response" in Fig. 2; imagine that the response is sent via traditional mail).

Some simplifying conventions (limiting the expressiveness of our tool) will be made. Contrary to [14, Section 10.3], here it will be assumed that every Task has at most (hence, exactly) one incoming Sequence Flow, at most one incoming Message Flow, and at most one outgoing Message Flow. Also, looping and multiplicity of Tasks will not be considered here.

Sub-Processes: BPMN defines some special types of Sub-Processes; here, however, we will only be interested in those that are just Processes within Processes (Embedded Sub-Processes). Examples can be seen in Fig. 2. Sub-Processes may have parallel multiplicity, i.e. multiple copies of a Sub-Process may run in parallel (looping or sequential multiplicity are also options in BPMN, but will not be considered here). Recall that here we require Sub-Processes to always contain at least one Start and one End Event. As for Tasks, contrary to [14, Section 10.3], here it will be assumed that every Sub-Process has at most (hence, exactly) one incoming Sequence Flow.

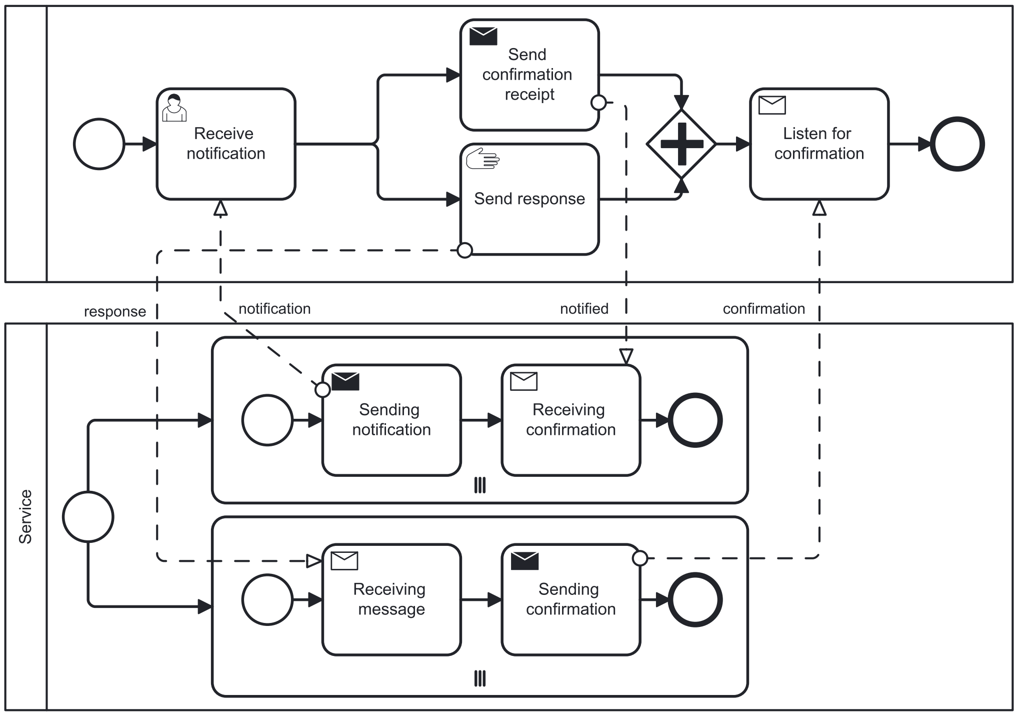


Figure 2: A BPMN diagram depicting a Collaboration. Here, the "Service" Pool has two Sub-Processes. These Sub-Processes have multiple (parallel) instances, indicated by the parallel lines at their bottom.

2.3 Flows

Flows are drawn as arrows. There are two kinds of flows: Sequence Flows (solid arrows) and Message Flows (dashed arrows).

Sequence Flows: Sequence Flows model the flow of control within a Process.

An Uncontrolled Flow (i.e. a Normal Flow not connected to some Gateway and not affected by Conditions) is the most basic kind of Flow, representing the order of execution of the elements it connects.

Conditional Flows (examples can be seen in Fig. 1b) are only activated if their corresponding Condition is met. Here, Conditional Flows will be considered only when they are outgoing from a Receive

Task (their Condition shall then pertain to the received message). Default Flows will not be considered.

Non-Normal Sequence Flows (pertaining to Exceptions and Compensations), as well as looping via "backwards" sequence flows, will not be considered here.

Of course, the restrictions on Flows heavily impact the expressiveness of the diagrams our tool can handle. However, the supported elements are already expressive enough to be considered in this version.

Message Flows: Message Flows depict the flow of messages between Participants in a Collaboration [14, Section 7.3.2] (see Fig. 2). According to [14, Chapter 10], "All Message Flows must connect two separate Pools. They may connect to the Pool boundary or to Flow Objects within the Pool boundary.". In this paper, each Message Flow must have a Flow Object as either source or target. Contrary to [14, Section 7.6.2], here it will be assumed that Sub-Processes per se have neither incoming nor outgoing Message Flows. Flow Nodes *within* a Sub-Process will support Message Flows as usual.

2.4 Gateways

Gateways control the convergence and divergence of Sequence Flows. Here, only Parallel and Exclusive Gateways will be considered (the latter only in their converging form). A Gateway must have multiple incoming Sequence Flows or multiple outgoing Sequence Flows (or multiple of both, which is not recommended) [14, Section 10.6]; recall that here it is also required to have at least one of each. "Gateways do not represent 'work' being done and they are considered to have zero effect on the operational measures of the Process being executed (cost, time, etc.)." [14, Section 10.6]

Parallel Gateways (e.g. the one synchronising the Tasks "Send confirmation receipt" and "Send response" in Fig. 2) create and synchronise parallel paths: that is, when multiple Flows are outgoing from a Parallel Gateway, all of them will be executed in parallel; dually, when multiple Flows are incoming to a Parallel Gateway, it will wait until all of them are executed before activating its output Flows.

According to [14, Section 10.6.2], "A converging Exclusive Gateway [i.e. one with multiple incoming Sequence Flows and one outgoing Sequence Flow] is used to merge alternative paths. Each incoming Sequence Flow token is routed to the outgoing Sequence Flow without synchronization.". Here, we will only consider cases where the incoming Sequence Flows to the Exclusive Gateway are alternative, i.e. that at most one of them can be triggered. An example of an Exclusive Gateway can be seen in Fig. 1b.

3 Privacy Calculus

The Privacy Calculus is a typed variant of the π-calculus introduced in [10] and further developed in [11, 9, 17, 16, 12, 23]. The version of the Privacy Calculus presented here is the one of [18], with the addition of the Choice and Silent operators which are standard in π-calculus and can be introduced in the tool of [18] with minimal effort (amounting to the addition of two transition rules, i.e. (Choice) and (Silent) of Fig. 3, and two simple typing rules).

Assume the following basic sets of entities: (1) an infinitely countable set of channel names (ranged over by x, y, z, a, b), (2) a set of basic types, (3) a set of purposes (ranged over by u), (4) a set of groups, (ranged over by G), split into a set of users, (ranged over by U) and a set of roles (ranged over by R); for any two groups G_1, G_2, their union $G_1 \cup G_2$ is also a group (notice that privacy policies support group hierarchies), (5) a set of context variables $\mathscr{X}$, where each $X \in \mathscr{X}$ has a finite domain D_X of possible values ranged over by v_X; v ranges over the union $\bigcup_X D_X$.

The following can then be defined: (1) a set of names $\mathscr{N}$ containing channel names and all the values of context variables, (2) a set of types $\mathscr{T}$ (ranged over by T), containing all basic types, all context variables, and every element of the form $G[T]$ (the intuition being that a name of type $G[T]$ can be used by processes that "belong" to group G in order to exchange messages of type T).

Terms of the Privacy Calculus are defined in two levels, processes and systems:

Process	P	::=	$\mathbf{0}$	Empty	System	S	::=	$\mathbf{0}$	Empty
		|	$x(x:T).P$	Input			|	$(\nu x:T)S$	Create channel
		|	$\overline{x}\langle x\rangle.P$	Output			|	$S \parallel S$	Parallel
		|	$\tau.P$	Silent			|	$R[S]$	Bind group
		|	$(\nu x:T)P$	Create channel			|	$G:u[P]$	Process lift
		|	$P \mid P$	Parallel					
		|	$P+P$	Choice					
		|	$[x=v](P;P)$	Conditional					
		|	$!P$	Replication					

Processes are standard in π-calculus: the empty process does nothing, the input $x(y:T).P$ receives y of type T via the channel x (binding the name y in P) and continues as P, the output $\overline{x}\langle y\rangle.P$ sends y via the channel x and continues as P, the silent process $\tau.P$ does some unspecified internal work and then continues as P, the process $(\nu x:T)P$ creates a channel x of type T (and binds the name x) in P, the parallel composition $P_1 \mid P_2$ combines the two processes so that both run in parallel, the choice composition P_1+P_2 combines the two processes so that only one will run, the conditional $[x=v](P_1;P_2)$ checks whether x is equal to v and if so continues as P_1, otherwise as P_2, and the replication $!P$ behaves as $P \mid !P$. For brevity, define $[x=v]P$ as $[x=v](P;\mathbf{0})$, $[x\neq v]P$ as $[x=v](\mathbf{0};P)$, $\prod_{i=1}^{n} P_i$ as $P_1 \mid \ldots \mid P_n$, and $\sum_{i=1}^{n} P_i$ as $P_1+\ldots+P_n$.

Systems annotate processes with high-level privacy information. The system $G:u[P]$ declares that the process P runs on behalf of group G for the purpose u, the system $R[S]$ declares that the system S runs for the role R (in addition to any other groups declared in S), while the empty, name binding, and parallel systems are similar to the respective processes. For brevity, define $\prod_{i=1}^{n} S_i$ as $S_1 \parallel \ldots \parallel S_n$.

For the unambiguous treatment of bound names, CINNI [21] is employed: it adds indices to names, so that, for example, the term $(\nu x:T)\overline{x}\langle y\rangle.(\nu x:T)\overline{x}\langle y\rangle.\mathbf{0}$ is actually interpreted as $(\nu x_1:T)\overline{x_1}\langle y\rangle.(\nu x_0:T)\overline{x_0}\langle y\rangle.\mathbf{0}$. Thus, name substitution $[a:=b]$ (substitute all free occurrences of a with b) can be defined elegantly. For technical reasons, CINNI defines some operators that convert indices, such that $[\texttt{shiftdown}\,aX]$, which decreases the indices of every a in X by 1 (not going below 0). In this paper, CINNI will be ignored, i.e. indices of channel names will always be omitted and the index 0 will be assumed for all channel names.

Two processes/systems that differ only in the selection of their bound names are called α-equivalent. Structural congruence (i.e. behavioural equivalence) of processes/systems, denoted $\equiv$, is defined as follows: (1) α-equivalent terms are congruent, (2) parallel/choice terms that differ only in the order of their operands are congruent (i.e. parallel and choice operators are associative and commutative), (3) repetitions of operands in choice is irrelevant (i.e. the choice operator is idempotent), (4) $P \mid \mathbf{0} \equiv P$, $S \parallel \mathbf{0} \equiv S$, $!\mathbf{0} \equiv \mathbf{0}$, $(\nu x:T)\mathbf{0} \equiv \mathbf{0}$ (both for processes and systems), $G[\mathbf{0}] \equiv \mathbf{0}$, $G:u[\mathbf{0}] \equiv \mathbf{0}$.

The operational semantics of the Privacy Calculus is defined in Fig. 3. It is a late labelled transition semantics comprising four kinds of labels: (1) silent, $\xrightarrow{\tau}$, (2) input, $\xrightarrow{x(y)}$, (3) output, $\xrightarrow{\overline{x}\langle y\rangle}$, (4) bound output $\xrightarrow{(\nu y:T)\overline{x}\langle y\rangle}$. Notice that the π-calculus, and hence Privacy Calculus, is non-deterministic: multiple execution steps might be possible for a given term, in which case any one of them might be selected arbitrarily as the next to be executed.

$$x(a:T).P \xrightarrow{x(a)} P \quad \text{(In)} \qquad \bar{x}\langle y\rangle.P \xrightarrow{\bar{x}\langle y\rangle} P \quad \text{(Out)} \qquad \tau.P \xrightarrow{\tau} P \quad \text{(Silent)}$$

$$\frac{P \xrightarrow{l} P'}{!P \xrightarrow{l} P' \mid !P} \quad \text{(Repl)} \qquad \frac{P_1 \xrightarrow{l} P_1'}{P_1 + P_2 \xrightarrow{l} P_1'} \quad \text{(Choice)}$$

$$\frac{P \xrightarrow{l} P'}{[x{=}x]\,(P\,;Q) \xrightarrow{l} P'} \quad \text{(CondT)} \qquad \frac{Q \xrightarrow{l} Q' \qquad x \neq y}{[x{=}y]\,(P\,;Q) \xrightarrow{l} Q'} \quad \text{(CondF)}$$

$$\frac{S \xrightarrow{l} S'}{R\,[S] \xrightarrow{l} R\,[S']} \quad \text{(ResGS)} \qquad \frac{P \xrightarrow{l} P'}{G : u\,[P] \xrightarrow{l} G : u\,[P']} \quad \text{(ResGP)}$$

$$\frac{F \xrightarrow{\bar{x}\langle a_0\rangle} F'}{(\nu\, a:T)F \xrightarrow{(\nu\, a_0:T)\bar{x}\langle a_0\rangle} F'} \quad \text{(Open)} \qquad \frac{F \xrightarrow{l} F' \qquad a_0 \notin \mathrm{fn}(l)}{(\nu\, a:T)F \xrightarrow{[\texttt{shiftdown}\, a\, l]} (\nu\, a:T)F'} \quad \text{(ResN)}$$

$$\frac{F_1 \xrightarrow{x(a)} F_1' \qquad F_2 \xrightarrow{\bar{x}\langle z\rangle} F_2'}{F_1 \mid F_2 \xrightarrow{\tau} ([a{:=}z]\,F_1') \mid F_2'} \quad \text{(Comm)} \qquad \frac{F_1 \xrightarrow{x(a)} F_1' \qquad F_2 \xrightarrow{(\nu\, b_n:T)\bar{x}\langle b_n\rangle} F_2'}{F_1 \mid F_2 \xrightarrow{\tau} (\nu\, b:T)(([a{:=}b_n]\,F_1') \mid F_2')} \quad \text{(Close)}$$

$$\frac{F_1 \xrightarrow{l} F_1' \qquad \mathrm{bn}(l) \cap \mathrm{fn}(F_2) = \emptyset}{F_1 \mid F_2 \xrightarrow{l} F_1' \mid F_2} \quad \text{(Par)} \qquad \frac{F_1 \equiv F_2 \qquad F_2 \xrightarrow{l} F}{F_1 \xrightarrow{l} F} \quad \text{(Congr)}$$

Figure 3: The rules of labelled transition semantics of the Privacy Calculus. $\mathrm{fn}(X)$ is the set of free names of the term X, while $\mathrm{bn}(X)$ is the set of its bound names. Rules that contain the variable F are applicable both to processes and systems.

4 Converting BPMN to Privacy Calculus

4.1 Main considerations regarding conversion

In the spirit of [19, 20, 1], a BPMN Process will be converted to a Privacy Calculus term consisting of the concatenation of terms corresponding to every Flow Node (i.e. Event, Activity, or Gateway) within the Process. Flows will be converted to channels that serve for communication between processes: Sequence Flows will carry tokens, while Message Flows will carry messages (possibly containing data important to privacy policies). Every Privacy Calculus process corresponding to a Flow Node will then have (roughly)[3] the following structure:

(1) begin with receiving tokens via its incoming Sequence Flows,

(2) continue with receiving Messages via its incoming Message Flows,

(3) do any work specific to its type,

[3]As [19, Section 4] points out, "The description given applies only to basic control flow structures. Advanced structures require slightly different approaches."; in fact, various of the patterns presented in [19, 20] and in Section 4.3 of this paper diverge (slightly or more radically) from the rough structure above. Moreover, "if a process representing [a Flow Node] can be triggered more than once, the replication operator must be used" and "a [conditional] prefix [after receiving the triggers] can be used to model global constraints like testing a cancellation flag" (the latter is in fact taken into account in the basic description given in [19], but is left out here).

(4) send messages via its outgoing Message Flows,

(5) pass token(s) to its outgoing Sequence Flow(s); some outgoing Sequence Flows might be affected by conditions, hence only be triggered conditionally.

This design choice is influenced by [19]: "A generic process can have m incoming triggers [...] and o outgoing triggers. [...] After the input prefixes have been triggered [...] First, the functional perspective of the activity is represented as an unobservable action. Second, the process can trigger other processes by output prefixes." [19, Section 4]. Here, we have "unfolded" a small part of the unobservable action so as to accommodate Message Flows and considered that some outgoing Sequence Flows are only conditionally triggered.

In order to accommodate Sequence Flows, the set $\mathscr{T}$ of types is presumed to contain a special type `Token` of tokens. For simplicity, values of this type will always be denoted by `t`, assuming that this name is not used for any other element.

Notice that "a token does not traverse a Message Flow since it is a Message that is passed down a Message Flow (as the name implies)" [14, Section 7.2]. Hence, for every Message Flow, the name and type (as an element of $\mathscr{T}$) of the Message needs to be known; types of Messages are not a part of BPMN and, as stressed in Section 2, names of Messages will need to be provided externally to BPMN.

4.2 Flow patterns

In [19, 20], various patterns common to business processes are identified and their conversion to π-calculus processes is discussed. We will review a few here (Sequence, Parallel split, Exclusive choice, Synchronisation, N–out–of–M–join) and present some variations of them (Choice, n–out–of–n synchronisation, m–out–of–n synchronisation).

While discussing flow patterns, the following notation will be adopted: (1) Instead of BPMN Flow Nodes and Flows, arbitrary nodes and edges (in the graph theoretic sense) will be considered. (2) The conversion of a node X into a Privacy Calculus term will be denoted $\|X\|$. (3) Since only the initial and/or final behaviour of X will be of interest, there will be a part of $\|X\|$ that will be irrelevant to this discussion (in fact, it will depend on what kind of BPMN element X is); this will be denoted by X'.

Sequence: The simplest pattern, defined in [19, Section 4.1].

Suppose that node A has a unique outgoing edge f. Then, A, when it has finished its work, needs only trigger the next node by sending a token via f, i.e. $\|A\| := A'.\overline{f}\langle \mathtt{t}\rangle.\mathbf{0}$.

Similarly, suppose that node B has a unique incoming edge f. Then, B waits until it receives the token and then starts its own work, i.e. $\|B\| := f(\mathtt{t} : \mathtt{Token}).B'$.

4.2.1 Outgoing

Suppose that node A has multiple outgoing edges $f_1, \ldots, f_n$ ($n \geq 2$) to other nodes.

Parallel split: In this pattern, defined in [19, Section 4.1], A triggers all of its outgoing edges in parallel. For $n = 2$, this can be achieved with $\|A\| := A'.(\overline{f_1}\langle \mathtt{t}\rangle.\mathbf{0} \mid \overline{f_2}\langle \mathtt{t}\rangle.\mathbf{0})$. This can be generalised to $\|A\| := A'.\prod_{i=1}^{n} \overline{f_i}\langle \mathtt{t}\rangle.\mathbf{0}$.

Exclusive choice: In this pattern, defined in [19, Section 4.1], A triggers exactly one of its outgoing edges. For $n = 2$, this can be achieved with $\|A\| := A'.(\overline{f_1}\langle \mathtt{t}\rangle.\mathbf{0} + \overline{f_2}\langle \mathtt{t}\rangle.\mathbf{0})$. This can be generalised to $\|A\| := A'.\sum_{i=1}^{n} \overline{f_i}\langle \mathtt{t}\rangle.\mathbf{0}$.

4.2.2 Incoming

Suppose that node B has multiple incoming edges $f_1,\ldots,f_n$ ($n \geq 2$) from other nodes.

Choice: In this pattern, B waits for any of its incoming edges to be triggered and then starts. Input from the rest of the edges is disregarded. For $n = 2$, this can be achieved with $\|B\| := f_1(\mathtt{t} : \mathtt{Token}).B' + f_2(\mathtt{t} : \mathtt{Token}).B'$. This can be generalised to $\sum_{i=1}^{n} f_i(\mathtt{t} : \mathtt{Token}).B'$.

If multiple incoming edges can be activated, then the choice pattern will process only one of them, leaving the rest "hanging". Depending on the situation at hand, this might be alleviated (if needed) either by creating a new copy of B for each incoming trigger (similarly to the *Multi-merge* pattern of [19, Section 4.2]) or by the *1-out-of-n synchronisation* pattern below.

Synchronisation: In this pattern, defined in [19, Section 4.1], B waits for all of its incoming edges to be triggered—in a predefined order, however—before it starts. For $n = 2$, this can be achieved with $\|B\| := f_1(\mathtt{t} : \mathtt{Token}).f_2(\mathtt{t} : \mathtt{Token}).B'$. This can be generalised to $f_1(\mathtt{t} : \mathtt{Token})\ldots.f_n(\mathtt{t} : \mathtt{Token}).B'$.

We will not use this pattern in Section 4.3, opting for the n-out-of-n and m-out-of-n variants below. The reason is twofold. First, notice that, in general, this pattern might create deadlock issues; for instance, consider A triggering both C and B (via f_2) and C triggering B (via f_1): if the outgoing pattern used by A waits for f_2 to be consumed before triggering C, then B will never be executed. Moreover, even if such deadlocks are guaranteed to be impossible, $\|B\|$ will be behaviourally different depending on the order the f_i are written; this asymmetry might be undesired in applications such as the one of Section 4.4 (e.g. it might complicate unit testing, since a single input will have multiple non-equivalent correct outputs).

***n*–out–of–*n* synchronisation:** In this pattern (similar to *N-out-of-M-join* of [19, Section 4.2]), B (run on behalf of group G) waits for all of its incoming edges to be triggered before it starts, consuming every trigger as it arrives. For $n = 2$, this can be achieved with

$$\|B\| := (\nu\, h : G[\mathtt{Token}])h(\mathtt{t} : \mathtt{Token}).h(\mathtt{t} : \mathtt{Token}).B' \mid f_1(\mathtt{t} : \mathtt{Token}).\overline{h}\langle\mathtt{t}\rangle.\mathbf{0} \mid f_2(\mathtt{t} : \mathtt{Token}).\overline{h}\langle\mathtt{t}\rangle.\mathbf{0},$$

where h must not be free in B'. This can be generalised to

$$\|B\| := (\nu\, h : G[\mathtt{Token}])\underbrace{h(\mathtt{t} : \mathtt{Token})\ldots.h(\mathtt{t} : \mathtt{Token})}_{n \text{ times}}.B' \mid \prod_{i=1}^{n} f_i(\mathtt{t} : \mathtt{Token}).\overline{h}\langle\mathtt{t}\rangle.\mathbf{0}\ .$$

The drawback of this pattern is that it creates a fresh name. Applications such as the one of Section 4.4 need to select a name not among the free names of B'. Moreover, it might complicate unit testing: either the selected name must be known when writing tests or α-equivalence must be tested instead of equality.

***m*–out–of–*n* synchronisation:** In this pattern, generalising the previous one, B (run on behalf of group G) waits for exactly $m \leq n$ of its incoming edges to be triggered before it starts, consuming however all n triggers as they arrive. For $n = 2$ and $m = 1$, this can be achieved with

$$\begin{aligned}\|B\| :=\,&(\nu\, h : G[\mathtt{Token}])(\nu\, r : G[\mathtt{Token}])\\ &r(\mathtt{t} : \mathtt{Token}).B' \mid f_1(\mathtt{t} : \mathtt{Token}).\overline{h}\langle\mathtt{t}\rangle.\mathbf{0} \mid f_2(\mathtt{t} : \mathtt{Token}).\overline{h}\langle\mathtt{t}\rangle.\mathbf{0} \mid h(\mathtt{t} : \mathtt{Token}).\overline{r}\langle\mathtt{t}\rangle.h(\mathtt{t} : \mathtt{Token}).\mathbf{0},\end{aligned}$$

where h and r must not be free in B'. This can be generalised to

$$\begin{aligned}\|B\| :=\,&(\nu\, h : G[\mathtt{Token}])(\nu\, r : G[\mathtt{Token}])r(\mathtt{t} : \mathtt{Token}).B' \mid \prod_{i=1}^{n} f_i(\mathtt{t} : \mathtt{Token}).\overline{h}\langle\mathtt{t}\rangle.\mathbf{0} \mid\\ &\underbrace{h(\mathtt{t} : \mathtt{Token})\ldots.h(\mathtt{t} : \mathtt{Token})}_{m \text{ times}}.\overline{r}\langle\mathtt{t}\rangle.\underbrace{h(\mathtt{t} : \mathtt{Token})\ldots.h(\mathtt{t} : \mathtt{Token})}_{n-m \text{ times}}.\mathbf{0}\ .\end{aligned}$$

The drawback of this pattern, as with the previous one, is that it creates fresh names. This pattern might be useful for the conversion of some kinds of Complex Gateways [14, Section 10.6.5] (e.g. those operating on the rule that "three out of five incoming Sequence Flows are needed to activate the Gateway"), which however are not considered here. [19, Section 4.2] defines the similar pattern *N–out–of–M–join*, which recursively restarts $\|B\|$ after having consumed all of the n input triggers.

4.3 Conversion of diagram elements

This section is the gist of this paper. In Sections 4.3.1 to 4.3.10, the conversion of every supported kind of BPMN element to Privacy Calculus is discussed. For any BPMN element N, the corresponding Privacy Calculus term will be denoted $\|N\|$.

4.3.1 Start Events

The most generic form of a Start Event N is for it to have (1) no incoming Sequence Flows, (2) multiple outgoing Sequence Flows $f_1, \dots, f_k$, $1 \le k$, and (3) (if it is a Message Start Event) multiple incoming Message Flows $E_1, \dots, E_l$, $1 \le l$, each carrying a message m_i of type T_i.

According to [14, Section 10.5.2], each Message Flow targeting a Start Event represents an instantiation mechanism (a trigger) for the Process; only one of the triggers is required to start a new Process. Thus, $\|N\|$ shall start with a *Choice* pattern among the Message Flows. Also, according to [14, Section 10.5.2], if multiple Sequence Flows originate at a Start Event, then they are considered as parallel paths; thus the *Parallel split* pattern shall be used. Hence, $\|N\|$ will be

$$\sum_{i=1}^{l} \left(E_i(m_i : T_i).(\nu\,\mathtt{t} : \mathtt{Token}) \prod_{j=1}^{k} \overline{f_j}\langle t \rangle.\mathbf{0} \right),$$

that is, N waits for any E_i to pass a message and then, being a Start Event, generates a token. It has no further internal work to do, so it triggers all of its outgoing Sequence Flows in parallel using the *Parallel split* pattern. Of course, in case there are no incoming Message Flows (i.e. the Start Event is not a Message Event), $\|N\|$ can be simplified to $(\nu\,\mathtt{t} : \mathtt{Token}) \prod_{j=1}^{k} \overline{f_j}\langle \mathtt{t} \rangle.\mathbf{0}$.

In case that a Message Start Event is part of a single Process (i.e. not in a Collaboration) or the modeller has failed to provide Message Flows, one "phantom" Message Flow can be assumed and the conversion can then still proceed as above.

4.3.2 End Events

The most generic form of an End Event N is for it to have (1) multiple incoming Sequence Flows $e_1, \dots, e_k$, $1 \le k$, (2) no outgoing Sequence Flows, and (3) (if it is a Message End Event) multiple outgoing Message Flows $F_1, \dots, F_l$, $1 \le l$, each carrying a message m_i of type T_i; it is assumed here that the messages are generated within the Event. Suppose that the Process containing N runs for group G.

Contrary to [14, Section 10.5.3], if multiple Sequence Flows converge into an End Event, they will be required to be parts of parallel paths; then, according to [14, Section 10.5.3], "the tokens will be consumed as they arrive". Hence, the End Event starts with a *k-out-of-k synchronisation* pattern. Afterwards, "Each Message Flow leaving the End Event will have a Message sent when the Event is triggered." [14, Section 10.5.3], which indicates a *Parallel split* of Message Flows. Hence, for $k > 1$, $\|N\|$ will be

$$(\nu\,h : G[\mathtt{Token}])\,\underbrace{h(\mathtt{t} : \mathtt{Token}).\dots.h(\mathtt{t} : \mathtt{Token})}_{k \text{ times}}.D \mid \prod_{i=1}^{k} e_i(\mathtt{t} : \mathtt{Token}).\overline{h}\langle \mathtt{t} \rangle.\mathbf{0},$$

where D is $\prod_{j=1}^{l}(\nu m_j : T_j)\overline{F_j}\langle m_j\rangle.\mathbf{0}$ for Message End Events and $\mathbf{0}$ otherwise. For $k = 1$, $\|N\|$ can be simplified to $e_1(\mathtt{t} : \mathtt{Token}).D$ (a *Sequence* pattern). If the End Event is part of a Sub-Process, the $\mathbf{0}$ at the end of D is replaced by a *Parallel split* pattern of the Sequence Flow(s) outgoing from the Sub-Process.

In case that a Message End Event is part of a single Process (i.e. not in a Collaboration) or the modeller has failed to provide Message Flows, one "phantom" Message Flow can be assumed and the conversion can then still proceed as above.

4.3.3 Intermediate Events

Recall that only Message Intermediate Events are considered in this paper.

Every Message Intermediate Event can be the source or target (depending on whether the Event is catching or throwing) of at most one Message Flow [14, Section 10.5.4]. Moreover, contrary to [14, Section 10.5.4], here it will be assumed that every Intermediate Event has at most (hence, exactly) one incoming Sequence Flow. The most generic form of a Message Intermediate Event N is hence for it to have (1) one incoming Sequence Flow e_1, (2) multiple outgoing Sequence Flows $f_1, \ldots, f_n$, $1 \le n$, and (3) (if it is a Message Intermediate Catch Event) one incoming Message Flow E, carrying a message m of type T, (4) (if it is a Message Intermediate Throw Event) one outgoing Message Flow F, carrying a message m of type T; it is assumed here that the outgoing message is generated within the Event.

According to [14, Section 10.5.4], if multiple Sequence Flows originate at an Intermediate Event, then they are considered as parallel paths. Hence the event can use the *Sequence* (for incoming) and *Parallel split* (for outgoing) patterns and $\|N\|$ is

$$e_1(\mathtt{t} : \mathtt{Token}).E(m : T).\prod_{j=1}^{n}\overline{f_i}\langle\mathtt{t}\rangle.\mathbf{0} \qquad \text{for Catch Events,}$$

$$e_1(\mathtt{t} : \mathtt{Token}).(\nu m : T)\overline{F}\langle m\rangle.\prod_{j=1}^{n}\overline{f_i}\langle\mathtt{t}\rangle.\mathbf{0} \qquad \text{for Throw Events.}$$

Notice that this is a simplified conversion. In fact, "if another token arrives from the same path or another path, then a separate instance of the Event will be created" [14, Section 10.5.4]. However, multiple instances of Events will not be tackled here, since that would be quite more complicated (as [19, Section 4.2] points out, "by using the replication operator to create multiple copies of a process D, all processes that are triggered by D must also support replication and so on. This also refers to all other patterns that create multiple copies by replication.") and of minimal interest regarding privacy protection.

In case that a Message Intermediate Event is part of a single Process (i.e. not in a Collaboration) or the modeller has failed to provide a Message Flow for the Event, one "phantom" Message Flow can be assumed and the conversion can then still proceed as above.

4.3.4 Parallel Gateways

Every Parallel Gateway N has (1) $1 \le k$ incoming Sequence Flows $e_1, \ldots, e_k$, and (2) $1 \le n$ outgoing Sequence Flows $f_1, \ldots, f_n$. Since Gateways have no internal operation, Parallel Gateways can be modelled using only the *k-out-of-k synchronisation* and *Parallel split* patterns, i.e. for $k > 1$, $\|N\|$ will be

$$(\nu h : G[\mathtt{Token}])\underbrace{h(\mathtt{t} : \mathtt{Token}).\ldots.h(\mathtt{t} : \mathtt{Token}).}_{k \text{ times}}\left(\prod_{i=1}^{n}\overline{f_i}\langle\mathtt{t}\rangle.\mathbf{0}\right) \mid \prod_{i=1}^{k}e_i(\mathtt{t} : \mathtt{Token}).\overline{h}\langle\mathtt{t}\rangle.\mathbf{0},$$

and for $k = 1$ it will be simplified to $e_1(\mathtt{t} : \mathtt{Token}).\left(\prod_{i=1}^{n}\overline{f_i}\langle\mathtt{t}\rangle.\mathbf{0}\right)$.

4.3.5 Exclusive Gateways

Under the simplifying conventions introduced in Section 2.4, an Exclusive Gateway N can only have the following form: (1) multiple incoming Sequence Flows $e_1, \ldots, e_k$, $2 \leq k$, at most one of which will be triggered, and (2) one outgoing Sequence Flow f_1. Hence, a *Choice* pattern for input and a *Sequence* pattern for output shall be adequate and $\|N\|$ can be $\sum_{i=1}^{k} e_i(\mathtt{t} : \mathtt{Token}).\overline{f_1}\langle \mathtt{t} \rangle.\mathbf{0}$.

4.3.6 Tasks

Under the simplifying conventions introduced in Sections 2.2 and 2.3, the most generic form of a Task N is for it to have (1) one incoming Sequence Flow e_1, (2) 0 or 1 incoming Message Flows E (1 in case the Task is a Receive Task), carrying a message m_E of type T_E, (3) multiple outgoing Sequence Flows $f_1, \ldots, f_n$, $1 \leq n$, where, if the Task is a Receive Task, each f_i might have a condition c_i attached (c_i compares m_E to some constant value v_i via o_i, where o_i can be either $=$ or $\neq$), (4) 0 or 1 outgoing Message Flows F (1 in case the Task is a Send Task), carrying a message m_F of type T_F; it is assumed here that the outgoing message is generated within the Task.

Outgoing Sequence Flows of Tasks need a *Parallel split* pattern, since "if there are multiple outgoing Sequence Flows, then this means that a separate parallel path is being created for each Sequence Flow" [14, Section 10.3]. For each other Flow kind, since at most one item exists, the *Sequence* pattern suffices.

In [19, Section 4], it is argued that "a process that represents an activity must have a functional part represented by τ"; recall that the $\tau.$ prefix in Privacy Calculus encodes that some unspecified internal work is performed. This is indeed compatible with the fact that Tasks are used in BPMN "when the work [...] is not broken down to a finer level of [...] detail" [14, Section 7.3.2].

Hence, $\|N\|$ will be

$$e_1(\mathtt{t} : \mathtt{Token}).E(m_E : T_E).\tau.(\nu\, m_F : T_F)\overline{F}\langle m_F \rangle.\prod_{i=1}^{n} [m_E\ o_i\ v_i]\overline{f_i}\langle \mathtt{t} \rangle.\mathbf{0}$$

where $E(m : T).$ shall be omitted if there is no E, $(\nu\, m : T)\overline{F}\langle m \rangle.$ shall be omitted if there is no F, and $[m_E\ o_i\ v_i]$ shall be omitted if there is no c_i. For a Send/Receive Task, the $\tau.$ shall be omitted, since there is no internal work other than sending/receiving the Message ([14, Section 10.3.3] stresses that once the Message has been sent/received, the Task is completed).

Notice that this is a simplified conversion. In fact, similarly to Intermediate Events, "if another token arrives from the same path or another path, then a separate instance of the Activity will be created" [14, Section 10.3]. This will not be considered here, with a same rationale as for Intermediate Events.

In case that a Receive/Send Task is part of a single Process (i.e. not in a Collaboration) or the modeller has failed to provide a Message Flow for the Task, one "phantom" Message Flow (incoming for Receive, outgoing for Send) can be assumed and the conversion can then still proceed as above.

4.3.7 Processes

As already mentioned in Section 4.1, a BPMN Process will be converted to a Privacy Calculus term consisting of the concatenation of Privacy Calculus subprocesses corresponding to every Flow Node within the Process. In fact, since we are interested in privacy protection, a top-level Process (i.e. not part of a Collaboration and not a Sub-Process) must be decorated with a group G that runs the Process and a purpose u for which it is run, so that it can be checked for compliance to privacy policies.

Notice that "each Start Event is an independent Event" [14, Section 10.5.2], hence in case of multiple Start Events in the same Process (something permitted but not recommended in BPMN [14, Section 10.5.2]), the first one to be triggered invalidates (for the Process instance that is created) the rest.

Consider a Process N. Let E_N be the set of Start Events of N. For every Start Event $E \in E_N$, let A_E be the set of Flow Nodes (including itself) that are accessible (in the graph-theoretic sense) via Sequence Flows from E. Let also S be the set of m Sequence Flows which connect the nodes of $\bigcup_{E \in E_N} A_E$; N must bind the names of the channels corresponding to the Flows in order to prevent usage from the outside.

Given the considerations above, $\|N\|$ will be the Privacy Calculus system

$$G : u \left[\underbrace{(\nu f_1 : G[\mathtt{Token}]) \ldots (\nu f_m : G[\mathtt{Token}])}_{\text{for all } f_i \in S, i=1,\ldots,m} \sum_{E \in E_N} \prod_{A \in A_E} \|A\| \right].$$

4.3.8 Sub-Processes

Under the conventions of Sections 2.2 and 2.3, the most generic form of a Sub-Process N is for it to have (1) one incoming Sequence Flow e_1, (2) multiple outgoing Sequence Flows $f_1, \ldots, f_n$, $1 \leq n$, (3) a non-empty set M of Flow Nodes and Flows within it.

Let F be the process $\prod_{i=1}^{n} \overline{f_i}\langle \mathtt{t} \rangle.\mathbf{0}$ (*Parallel split* of the outgoing Sequence Flows). Then $\|N\|$ will be $e_1(\mathtt{t} : \mathtt{Token}).\|M\|$, where M is converted as a Process (Section 4.3.7), with the exceptions that (1) as noted in Section 4.3.2, End Events of Sub-Processes are converted in a special manner: as their final step, instead of a plain $\mathbf{0}$, they contain F, thus shifting flow control back to the process that contains N, (2) a Sub-Process is not decorated with group/purpose information: it is considered to run for the same group and purpose as the Process containing it. If N is a multi-instance (parallel) Sub-Process, then $\|N\|$ is $e_1(\mathtt{t} : \mathtt{Token}).!\,\|M\|$.

4.3.9 Participants

A Participant N has, in general, one of the two following structures:

- It is either a Process M with the (optional) information of a group/user G that runs it. If we make this information required and also require a purpose u, then $\|N\|$ can be the Privacy Calculus system $G : u[\|M\|]$,
- Or it is just a group/user G (again, the name is optional) depicted as a "black box". In this case, we can use a variable P_G for the Privacy Calculus process, require a group and a purpose, and set $\|N\|$ to be the Privacy Calculus system $G : u[P_G]$.

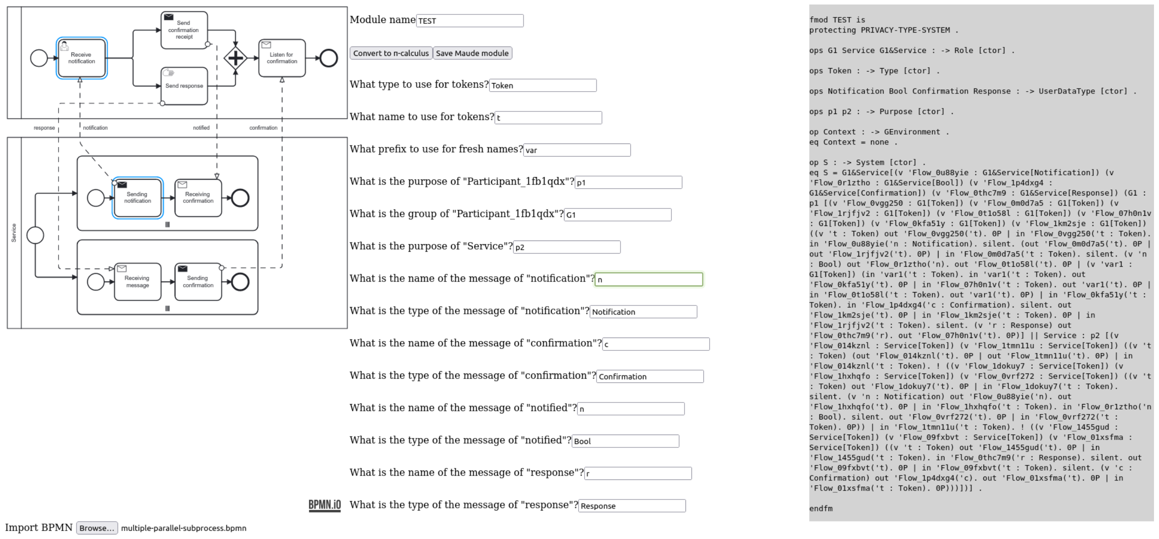
4.3.10 Collaborations

A Collaboration N is a non-empty collection of Participants $M_1, \ldots, M_n$, $1 \leq n$, of groups $G_1, \ldots G_n$, along with some Message Flows $F_1, \ldots, F_k$, $0 \leq k$, each F_i carrying a Message of type T_i between two Participants pertaining to groups $G_{i,1}$ and $G_{i,2}$. In addition to containing Participants, the converted term of the Collaboration needs to bind the Message Flows (for exactly the same reasons as Process binds Sequence Flows) and declare the group combinations in use. Thus, $\|N\|$ is

$$G_{1,1} \cup G_{1,2} \left[\ldots G_{k,1} \cup G_{k,2} \left[(\nu F_1 : G_{1,1} \cup G_{1,2}[T_1]) \ldots (\nu F_k : G_{k,1} \cup G_{k,2}[T_k]) \prod_{i=1}^{n} \|M_i\| \right] \right].$$

4.4 A tool that automates the conversion

An open source tool that automates the conversion has been implemented and its source code is available at [15]. It is a simple web application written in HTML and Javascript (ES2022 dialect), which can run in modern web browsers. A screenshot of the app in use is shown in Fig. 4.



multiple-parallel-subprocess.bpmn

Module name TEST

What type to use for tokens? Token

What name to use for tokens? t

What prefix to use for fresh names? var

What is the purpose of "Participant_1fb1qdx"? p1

What is the group of "Participant_1fb1qdx"? G1

What is the purpose of "Service"? p2

What is the name of the message of "notification"? n

What is the type of the message of "notification"? Notification

What is the name of the message of "confirmation"? c

What is the type of the message of "confirmation"? Confirmation

What is the name of the message of "notified"? n

What is the type of the message of "notified"? Bool

What is the name of the message of "response"? r

What is the type of the message of "response"? Response

```
fmod TEST is
protecting PRIVACY-TYPE-SYSTEM .

ops G1 Service G1&Service : -> Role [ctor] .

ops Token : -> Type [ctor] .

ops Notification Bool Confirmation Response : -> UserDataType [ctor] .

ops p1 p2 : -> Purpose [ctor] .

op Context : -> GEnvironment .
eq Context = none .

op S : -> System [ctor] .
eq S = G1&Service[(v 'Flow_0u88yie : G1&Service[Notification]) (v
'Flow_0r1ztho : G1&Service[Bool]) (v 'Flow_1p4dxg4 :
G1&Service[Confirmation]) (v 'Flow_0thc7m9 : G1&Service[Response]) (G1 :
p1 [(v 'Flow_0vgg250 : G1[Token]) (v 'Flow_0m0d7a5 : G1[Token]) (v
'Flow_1rjfjv2 : G1[Token]) (v 'Flow_0t1o58l : G1[Token]) (v 'Flow_07h0n1v
: G1[Token]) (v 'Flow_0kfa51y : G1[Token]) (v 'Flow_1km2sje : G1[Token])
((v 't : Token) out 'Flow_0vgg250('t). 0P | in 'Flow_0vgg250('t : Token).
in 'Flow_0u88yie('n : Notification). silent. (out 'Flow_0m0d7a5('t). 0P |
out 'Flow_1rjfjv2('t). 0P) | in 'Flow_0m0d7a5('t : Token). silent. (v 'n
: Bool) out 'Flow_0r1ztho('n). out 'Flow_0t1o58l('t). 0P | (v 'var1 :
G1[Token]) (in 'var1('t : Token). in 'var1('t : Token). out
'Flow_0kfa51y('t). 0P | in 'Flow_07h0n1v('t : Token). out 'var1('t). 0P |
in 'Flow_0t1o58l('t : Token). out 'var1('t). 0P) | in 'Flow_0kfa51y('t :
Token). in 'Flow_1p4dxg4('c : Confirmation). silent. out
'Flow_1km2sje('t). 0P | in 'Flow_1km2sje('t : Token). 0P | in
'Flow_1rjfjv2('t : Token). silent. (v 'r : Response) out
'Flow_0thc7m9('r). out 'Flow_07h0n1v('t). 0P)] || Service : p2 [(v
'Flow_014kznl : Service[Token]) (v 'Flow_1tmn11u : Service[Token]) ((v 't
: Token) (out 'Flow_014kznl('t). 0P | out 'Flow_1tmn11u('t). 0P) | in
'Flow_014kznl('t : Token). ! ((v 'Flow_1dokuy7 : Service[Token]) (v
'Flow_1hxhqfo : Service[Token]) (v 'Flow_0vrf272 : Service[Token]) ((v 't
: Token) out 'Flow_1dokuy7('t). 0P | in 'Flow_1dokuy7('t : Token).
silent. (v 'n : Notification) out 'Flow_0u88yie('n). out
'Flow_1hxhqfo('t). 0P | in 'Flow_1hxhqfo('t : Token). in 'Flow_0r1ztho('n
: Bool). silent. out 'Flow_0vrf272('t). 0P | in 'Flow_0vrf272('t :
Token). 0P)) | in 'Flow_1tmn11u('t : Token). ! ((v 'Flow_1455gud :
Service[Token]) (v 'Flow_09fxbvt : Service[Token]) (v 'Flow_01xsfma :
Service[Token]) ((v 't : Token) out 'Flow_1455gud('t). 0P | in
'Flow_1455gud('t : Token). in 'Flow_0thc7m9('r : Response). silent. out
'Flow_09fxbvt('t). 0P | in 'Flow_09fxbvt('t : Token). silent. (v 'c :
Confirmation) out 'Flow_1p4dxg4('c). out 'Flow_01xsfma('t). 0P | in
'Flow_01xsfma('t : Token). 0P)))])] .

endfm
```

Figure 4: A screenshot of the web app presented in Section 4.4. At the left, the imported diagram is shown and a new one can be uploaded. The middle part contains the extra information that the user needs to fill; for ease of use, the relevant Flow Nodes are highlighted when the user selects a question. The right part contains the Maude module created by the app (or the latest error that occurred).

First, a BPMN XML diagram is imported. It must adhere to the assumptions mentioned in this paper. The tool uses the open source library `bpmn-js` to parse BPMN XML diagrams[4].

Afterwards, the app asks for any extra info needed. It always requests the names to be used as (1) the type of tokens, (2) the value of tokens, (3) prefix of fresh names; this avoids using predefined values. Moreover, it asks for purposes and missing groups of Processes/Participants, and details (name, type, and, if "phantom", channel) of Messages carried by (actual or "phantom") Message Flows.

Finally, a Maude module compatible with [18] is created. It contains a Privacy Calculus system `S` built using the conversions defined in Section 4. In order for Maude to parse it correctly, all groups, types, purposes, values of user data types, and process/system variables used in `S` are first defined as terms of the module. For technical reasons having to do with the type checking algorithm of Privacy Calculus, the module also contains the context of `S`, i.e. a formal term specifying the types of free names of `S`. The Maude module can then be saved to a file and imported in the tool of [18] for further processing.

5 Conclusion and future work

We have presented how some basic elements of BPMN diagrams can be converted into Privacy Calculus and have provided a tool which can perform the conversion automatically and export it in a form compatible with the Maude formalisation of Privacy Calculus in [18].

[4]The authors of `bpmn-js` have also created the app bpmn.io that can be used for creating and editing BPMN XML diagrams.

As detailed in the previous sections, a significant subset of BPMN was not considered in this paper (and in the tool), but is required for detailed modelling of business processes. It is a matter of future work to integrate more aspects of BPMN. For instance, supporting BPMN Data Objects and Data Stores would significantly boost the level of expressiveness; this can be achieved by taking advantage of some versions of Privacy Calculus that contain operators for data storage and retrieval [12, 22].

A more accurate conversion of some Events and Activities could be achieved by carefully converting Flow Nodes to account for re-triggering. This might not be quite important *as far as privacy is concerned*[5], since a business process is compliant iff every path is compliant, hence re-triggering a path or triggering another one is irrelevant. Of course, sound conversion[6] is important for other reasons.

The toolchain of Privacy Calculus can (and should) also be complemented with more tools. For instance, a workflow for proving compliance of programs to policies would require tools that aid in the declaration of privacy policies, either by offering GUIs for the policy language of Privacy Calculus or by converting from more user-friendly frameworks.

References

[1] Riad Boussetoua, Hammadi Bennoui, Allaoua Chaoui, Khaled Khalfaoui & Elhillali Kerkouche (2015): *An automatic approach to transform BPMN models to Pi-Calculus*. In: *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, pp. 1–8, doi:10.1109/AICCSA.2015.7507176. ISSN: 2161-5330.

[2] Bert de Brock (2024): *Assigning Declarative Semantics to Some UML Activity Diagrams and BPMN Diagrams*. In Boris Shishkov, editor: *Business Modeling and Software Design*, Springer Nature Switzerland, Cham, pp. 65–82, doi:10.1007/978-3-031-64073-5_5.

[3] Egon Börger & Ove Sörensen (2011): *BPMN Core Modeling Concepts: Inheritance-Based Execution Semantics*. In David W. Embley & Bernhard Thalheim, editors: *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, Springer, Berlin, Heidelberg, pp. 287–332, doi:10.1007/978-3-642-15865-0_9.

[4] Flavio Corradini, Chiara Muzi, Barbara Re, Lorenzo Rossi & Francesco Tiezzi (2022): *BPMN 2.0 OR-Join Semantics: Global and local characterisation*. *Information Systems* 105, p. 101934, doi:10.1016/j.is.2021.101934.

[5] Flavio Corradini, Andrea Polini, Barbara Re & Francesco Tiezzi (2016): *An Operational Semantics of BPMN Collaboration*. In Christiano Braga & Peter Csaba Ölveczky, editors: *Formal Aspects of Component Software*, Springer International Publishing, Cham, pp. 161–180, doi:10.1007/978-3-319-28934-2_9.

[6] Remco Dijkman & Pieter Van Gorp (2010): *BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules*. In Jan Mendling, Matthias Weidlich & Mathias Weske, editors: *Business Process Modeling Notation*, Springer, Berlin, Heidelberg, pp. 16–30, doi:10.1007/978-3-642-16298-5_4.

[7] Remco M. Dijkman, Marlon Dumas & Chun Ouyang (2008): *Semantics and analysis of business process models in BPMN*. *Information and Software Technology* 50(12), pp. 1281–1294, doi:10.1016/j.infsof.2008.02.006.

[8] Outman El Hichami, Mohamed Naoum, Mohammed Al Achhab, Ismail Berrada & Badr Eddine El Mohajir (2015): *Towards a Formal Semantics and Analysis of BPMN Gateways*. In Ahmed Bouajjani &

[5]At least at the current level of maturity of the Privacy Calculus ecosystem: if policies became more expressive (e.g. if they placed restrictions on *how much* or *how often* data is processed), then consideration of re-triggering would be required.

[6]The BPMN standard does not define a formal semantics for diagrams, resting only on the treatment of tokens. Multiple researchers have proposed formalisations of (parts of) BPMN semantics in different frameworks: a few are [7, 6, 3, 24, 13, 8, 5, 4, 2]. Hence, soundness is in general not uniquely determined, since it depends on the selected formalisation.

Hugues Fauconnier, editors: *Networked Systems*, Springer International Publishing, Cham, pp. 474–478, doi:10.1007/978-3-319-26850-7_34.

[9] Eleni Kokkinofta & Anna Philippou (2014): *Type Checking Purpose-Based Privacy Policies in the π-calculus*. In: *Web Services, Formal Methods, and Behavioral Types*, Lecture Notes in Computer Science, Springer, Cham, pp. 122–142, doi:10.1007/978-3-319-33612-1_8.

[10] Dimitrios Kouzapas & Anna Philippou (2014): *A Typing System for Privacy*. In Steve Counsell & Manuel Núñez, editors: *Software Engineering and Formal Methods*, Springer International Publishing, Cham, pp. 56–68, doi:10.1007/978-3-319-05032-4_5.

[11] Dimitrios Kouzapas & Anna Philippou (2015): *Type Checking Privacy Policies in the π-calculus*. In: *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, Springer, Cham, pp. 181–195, doi:10.1007/978-3-319-19195-9_12.

[12] Dimitrios Kouzapas & Anna Philippou (2017): *Privacy by typing in the π-calculus*. *Logical Methods in Computer Science* 13(4), doi:10.23638/LMCS-13(4:27)2017.

[13] Vitus Lam (2012): *A Precise Execution Semantics for BPMN*. *IAENG International Journal of Computer Science* 39.

[14] Object Management Group (2013): *Business Process Model and Notation*. Available at `https://www.omg.org/spec/BPMN/2.0.2`. Version 2.0.2.

[15] Georgios V. Pitsiladis: *BPMN to Privacy Calculus converter*, doi:10.5281/zenodo.13235352.

[16] Georgios V. Pitsiladis (2016): *Type checking conditional purpose-based privacy policies in the π-calculus*. In: *1st Workshop for Formal Methods on Privacy*, Limassol, Cyprus. Available at `https://easychair.org/publications/preprint/Nnd7`.

[17] Georgios V. Pitsiladis (2016): *Type checking privacy policies in the π-calculus and its executable implementation in Maude*. Diploma thesis (in Greek), National Technical University of Athens, Athens, Greece, doi:10.26240/heal.ntua.10994.

[18] Georgios V. Pitsiladis & Petros Stefaneas (2018): *Implementation of Privacy Calculus and Its Type Checking in Maude*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 477–493, doi:10.1007/978-3-030-03421-4_30.

[19] Frank Puhlmann & Mathias Weske (2005): *Using the π-Calculus for Formalizing Workflow Patterns*. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati & Francisco Curbera, editors: *Business Process Management*, Springer, Berlin, Heidelberg, pp. 153–168, doi:10.1007/11538394_11.

[20] Frank Puhlmann & Mathias Weske (2006): *Investigations on Soundness Regarding Lazy Activities*. In Schahram Dustdar, José Luiz Fiadeiro & Amit P. Sheth, editors: *Business Process Management*, Springer, Berlin, Heidelberg, pp. 145–160, doi:10.1007/11841760_11.

[21] Mark-Oliver Stehr (2000): *CINNI - A Generic Calculus of Explicit Substitutions and its Application to λ-ς- and π-calculi*. *Electronic Notes in Theoretical Computer Science* 36, pp. 70–92, doi:10.1016/S1571-0661(05)80125-2.

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

[23] Evangelia Vanezi, Dimitrios Kouzapas, Georgia M. Kapitsaki & Anna Philippou (2020): *Towards GDPR Compliant Software Design: A Formal Framework for Analyzing System Models*. In Ernesto Damiani, George Spanoudakis & Leszek A. Maciaszek, editors: *Evaluation of Novel Approaches to Software Engineering*, Communications in Computer and Information Science, Springer International Publishing, Cham, pp. 135–162, doi:10.1007/978-3-030-40223-5_7.

[24] Peter Y. H. Wong & Jeremy Gibbons (2011): *Formalisations and applications of BPMN*. *Science of Computer Programming* 76(8), pp. 633–650, doi:10.1016/j.scico.2009.09.010.

Efficient Performance Analysis of Modular Rewritable Petri Nets

Lorenzo Capra
Dipartimento di Informatica
Università degli Studi di Milano, Italy

Marco Gribaudo
Dipartimento di Elettronica, Informatica e Bioingeneria
Politecnico di Milano, Italy

Petri Nets (PN) are extensively used as a robust formalism to model concurrent and distributed systems; however, they encounter difficulties in accurately modeling adaptive systems. To address this issue, we defined rewritable PT nets (RwPT) using `Maude`, a declarative language that ensures consistent rewriting logic semantics. Recently, we proposed a modular approach that employs algebraic operators to build extensive RwPT models. This methodology uses composite node labeling to maintain hierarchical organization through net rewrites and has been shown to be effective. Once stochastic parameters are integrated into the formalism, we introduce an automated procedure to derive a *lumped* CTMC from the quotient graph generated by a modular RwPT model. To demonstrate the effectiveness of our method, we present a fault-tolerant manufacturing system as a case study.

1 Introduction

Despite their potential, traditional formalisms such as Petri Nets, Automata, and Process Algebra do not easily allow designers to define dynamic changes in systems or assess their performance impact. As a result, many extensions to these classical models have been proposed, such as the π-calculus and the Nets-within-Nets paradigm, although they often lack adequate analysis techniques.

Rewritable PT nets (RwPT) were introduced in [7] as a versatile formalism for the modeling and analysis of adaptive distributed systems. The RwPT procedures were defined using the declarative language `Maude`, which leverages Rewriting Logic to offer both operational and mathematical semantics, thereby enabling a scalable framework for self-adapting PT nets. Unlike comparable methods ([2, 12]), which convert a simpler type of PNs into `Maude`, the RwPT formalism simplifies data abstraction, is concise and effective, and circumvents the limitations imposed by the pushout mechanism common in Graph Transformation Systems. RwPT extends GTS. It is vital to consider graph isomorphism (GI) in recognizing equivalent states within the model dynamics. This consideration is particularly advantageous for scaling up the model's size or degree of parallelism, especially when integrating a Stochastic Process into the model's state space. Recent research has demonstrated that GI has a quasi-polynomial complexity [1]. Graph canonization (GC), which is at least as complex as GI, involves determining a canonical form for any graph such that for any two graphs G and G', $G \simeq G' \Leftrightarrow can(G) = can(G')$. We have developed a general canonization method [6] for use with RwPT, integrated into `Maude`. This method is effective for irregular models, but it is less efficient for more realistic models that contain numerous similar components organized in a nested structure.

In [8], we introduced a method for developing extensive RwPT models using algebraic operators. Our approach is simple: Exploiting the modular features of the models during analysis. Using composite node labeling, we identify symmetries and maintain hierarchical organization through net rewrites. A case study (used as benchmark) demonstrates the success of our method, showing performance benefits over somewhat related approaches. In this paper, we present an automated technique to derive a Lumped

Continuous-Time Markov Chain (CTMC) from the quotient graph generated by an RwPT model after embedding stochastic parameters into the framework.

Background information is provided in Section 2, and our example is described in Section 3. The modular RwPT formalism, now with stochastic parameters, is explained in Section 4. In Section 5, we detail the method for deriving a lumped CTMC from an RwPT model and present experimental evidence of its effectiveness. We conclude by discussing ongoing work.

2 Background: (Stochastic) PT Nets and `Maude`

This section provides a concise overview of the (stochastic) PT formalism and emphasizes the key aspects of the `Maude` framework. For exhaustive information, we direct readers to the reference papers.

A *multiset* (*bag*) b in a nonempty set D is a map $b : D \to \mathbb{N}$, where $b(d)$ is the *multiplicity* of d in b. A multiset is empty if all of its multiplicities are zero. We denote by $Bag[D]$ the set of multisets in D. Standard relational and arithmetic operations can be applied to multisets on a component-by-component basis. In particular, let $b, b' \in Bag[D]$:

$b + b' \in Bag[D]$ is $b + b'(d) = b(d) + b'(d), \forall d \in D$.

$b < b' = true \Leftrightarrow \forall d \in D\ b(d) < b'(d)$.

$b - b' \in Bag[D]$ is defined if $b' \leq b$: $b - b'(d) = b(d) - b'(d), \forall d \in D$.

A stochastic PT (or SPN) *net* [13, 10] is a 6-tuple (P, T, I, O, H, λ), where: P, T are finite, non-empty, disjoint sets holding the net's nodes (places and transitions, respectively); $I, O, H : T \to Bag[P]$ represent the transitions' *input*, *output*, and *inhibitor* incidence matrices, respectively; $\lambda : T \to \mathbb{R}^+$ assigns each transition a negative exponential firing rate. A PT net *marking* is a multiset $m \in Bag[P]$.

The PT net dynamics is defined by the *firing rule*: $t \in T$ is *enabled* in marking m if and only if:

$$I(t) \leq m \wedge \forall p \in P :\ H(t)(p) > 0 \Rightarrow m(p) < H(t)(p)$$

If t is enabled m it may fire, leading to marking

$$m' = m - I(t) + O(t)$$

The notation $m[t\rangle m'$ means that t is enabled in m and its firing leads to m'.

A PT-*system* is a pair (N, m_0), where N is a PT net and m_0 is a marking of N. The interleaving semantics of (N, m_0) is specified by the *reachability graph* (RG): the RG is an edge-labelled, directed graph (V, E) whose nodes are markings. It is defined inductively: $m_0 \in V$; if $m \in V$ and $m[t\rangle m'$ then $m' \in V$, $m \xrightarrow{t} m' \in E$.

The timed semantics of a stochastic PT system is a CTMC isomorphic to the RG. For any two $m_i, m_j \in V$, the transition rate from m_i to m_j is $r_{i,j} := \sum_{t: m_i[t\rangle m_j} \lambda(t)$. The CTMC infinitesimal generator is a $|V| \times |V|$ matrix Q such that $Q[i,j] = r_{i,j}$ if $i \neq j$, $Q[i,i] = 1 - \sum_{j, j \neq i} r_{i,j}$.

In Generalized Stochastic Petri Nets (GSPN) [10] transitions can be assigned a priority (the firing rule is extended accordingly): transitions with a priority greater than zero occur instantly, and the associated stochastic parameters (denoted by the function λ) are used to resolve potential conflicts probabilistically. Consequently, their timed semantics leads to a Continuous-Time Markov Chain (CTMC) that is isomorphic to the "reduced" RG, obtained by eliminating nonobservable markings. This paper focuses on Stochastic Petri Nets (SPN) even though our specification encompasses GSPN.

The `Maude` **system** `Maude` [11] is a highly expressive, purely declarative language characterized by a rewriting logic semantics [4]. Statements consist of (conditional) *equations* and *rules*. Each side of a rule or equation represents terms of a specific *kind* that might include variables. The semantics of rules and equations involve straightforward rewriting, where instances of the left-hand side are substituted by corresponding instances of the right-hand side. The expressivity of `Maude` is realized through the use of matching modulo operator equational attributes, sub-typing, partiality, generic types, and reflection. A `Maude` *functional* module comprises only *equations* and functions as a functional program defining one or more operations through equations, utilized as simplification rules. A functional module details an *equational theory* within membership equational logic [3]. Formally, such a theory is a tuple $(\Sigma, E \cup A)$, with Σ representing the signature, which includes the declaration of all sorts, subsorts, kinds[1], and operators; E being the set of equations and membership axioms; and A as the set of operator equational attributes (e.g., `assoc`). The model of $(\Sigma, E \cup A)$ is the *initial algebra* $T_{\Sigma/E\cup A}$, which mathematically corresponds to the quotient of the ground-term algebra T_Σ. Provided that E and A satisfy nonrestrictive conditions, the final (or *canonical*) values of ground terms form an algebra isomorphic to the initial algebra, ensuring that the mathematical and rewriting semantics are identical.

A `Maude` *system module* includes *rewrite rules* and, potentially, equations. These rules illustrate local transitions in a concurrent system. In formal language, a system module outlines a generalized *rewrite theory* [4], symbolized as a four-tuple $\mathscr{R} = (\Sigma, E \cup A, \phi, R)$, where $(\Sigma, E \cup A)$ constitutes a membership equational theory; ϕ identifies the frozen arguments for each operator in Σ; and R contains a set of rewrite rules [2]. This rewrite theory models a concurrent system. $(\Sigma, E \cup A)$ establishes the algebraic structure of the states, while R and ϕ define the concurrent transitions of the system. The initial model of $\mathscr{R}$ assigns to each kind k a labeled transition system (TS) where the states are the elements of $T_{\Sigma/E\cup A,k}$, and transitions occur as $[t] \overset{[\alpha]}{\to} [t']$, with $[\alpha]$ representing *equivalent* rewrites. The property of *coherence* guarantees that a strategy that reduces terms to their canonical forms before applying the rules is sound and complete. A `Maude` system module is also an executable specification of distributed systems. Given finite reachability, it enables the verification of invariant properties and the discovery of counterexamples. Moreover, it supports the verification of LTL formulas. When the TS generated by a ground term becomes excessively large or infinite, bounded searches or abstractions might be employed.

3 Gracefully Degrading Production System

The illustrative example in this paper depicts a distributed production system that degrades gracefully, whose base configuration is shown by the two PT systems in Figure 1. The upper net represents a Production Line (denoted PL) which is divided into K branches (robots) that handle raw materials (a multiple of K). These branches ($\{w_i, ln_i, a_i\}$, $i : 0 \ldots K-1$) are fully interchangeable. An assembly component (transition *as*) converts the processed materials K into an artifact. A loader (ld) collects K items from a storage facility (place s) on the K lines of the PL. In this study, $K = 2$. The initial count of pieces (tokens) in s is $K \cdot M$, where $M \in \mathbb{N}^+$ is another parameter of the model. For each artifact produced, fresh items K are introduced. A branch might fail (transitions ft_i). When that occurs, the PL restructures to continue functioning, but with reduced capacity. Simple static analysis can show that the PL system reaches a *deadlock* after a failure.

[1]Kinds are implicit equivalence classes defined by connected components of sorts (as per subsort partial order). Terms in a kind without a specific sort are *error* terms.

[2]Rewrite rules do not apply to frozen arguments.

The net at the bottom of Figure 1 shows the transformation of the PL after a fault happens (considering scenario $K = 2$). This process involves moving items from the faulted branch to the remaining branch(es) to maintain the production cycle. Traditional PN frameworks (including High-Level PN variants) are unable to model this operation. Items left on the faulty line (represented as place w_1 here) are transferred to the remaining functional line (w_0): The marking of the PT net at the bottom demonstrates the state after adaptation. We assume that a PL that fails twice is beyond repair.

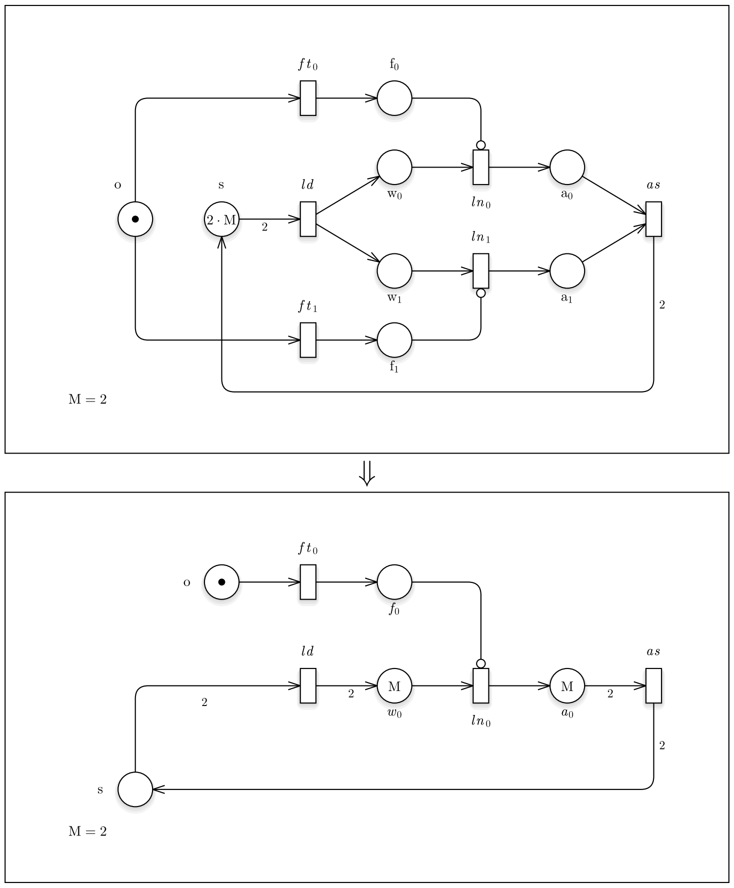


Figure 1: Production Line (PL) and adaptation following a fault.

We will examine a more complex scenario in which N PL replicas function simultaneously and degrade in a regulated fashion. Figure 2 demonstrates one potential evolution of a system starting with two PLs: This scenario can be extended to a system that incorporates N PLs, each operating K parallel robots, that handles $K \cdot M$ raw items, denoted by the term `NPLsys(N, K, M)`. The graceful degradation of the system proceeds in two phases:

s1 When a fault impacts a robot (line) of a PL, the PL autonomously adjusts to continue functioning in a diminished capacity (for simplicity, we here consider a two-lines scenario)

s2 When a second fault occurs in a degraded PL, the PL is disconnected from the entire system (see the final step in Figure 2): The leftover items are then relocated to the warehouse.

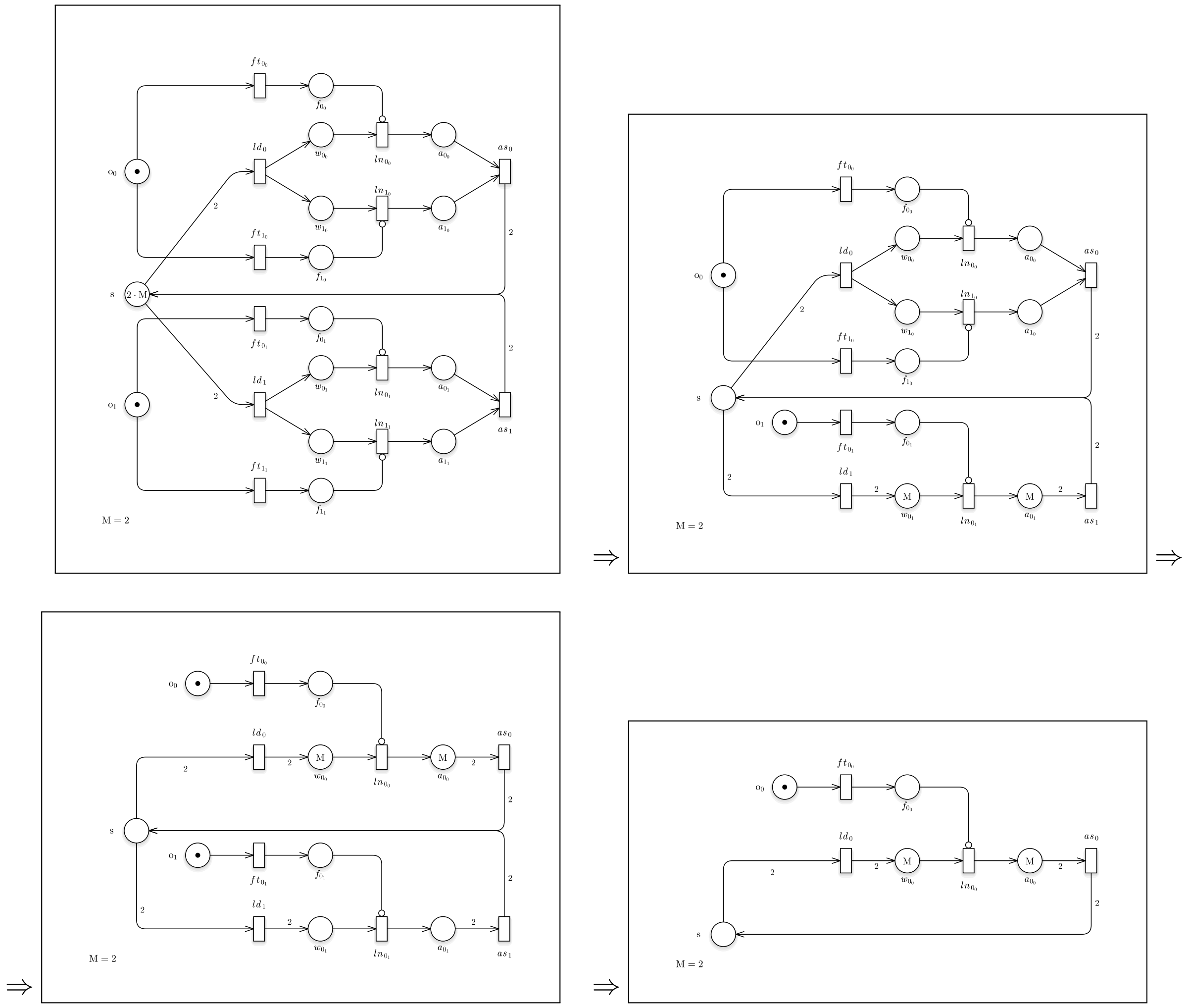


Figure 2: One of the possible paths of the Gracefully Degrading Production System.

4 Modular Rewritable Stochastic PN

This section introduces the concept of rewritable stochastic PT nets (RwSPT). These expand on the *modular* rewritable PT nets described in [8] by incorporating transitions with priorities and stochastic parameters. An RwSPT serves as an algebraic model of a Generalized Stochastic PN [10], combining rewrite rules with the PT firing mechanism. In this study, we concentrate on stochastic PN consisting of zero-priority transitions accompanied by an exponential firing rate.

The definition of RwSPT includes a hierarchy of **Maude** modules (e.g., `BAG`, `PT-NET`, `PT-SYSTEM`) most of which described in [8]. The `Maude` sources can be found in https://github.com/lgcapra/rewpt/tree/main/modSPT.

The RwSPT definition uses structured annotations to underline the model's symmetry. It features a concise place-based encoding to aid in state canonization and is based on the functional module `BAG{X}`, which introduces multisets as a complex data type. Specifically, the commutative and associative `_+_` operator provides an intuitive way to describe a multiset as a weighted sum, for instance, `3 . a + 1 . b`. The sort `Pbag` contains multisets of places.

Each place label (a term of sort `Plab`) is a non-empty list of pairs built of `String` and a `Nat`. Places are uniquely identified by their labels. These pairs represent a symmetric component within a nested hierarchy. Compositional operators annotate places incrementally from right to left: The label suffix represents the root of a hierarchy. For example, the 'assembly' place of line 1 in Production Line 2 would be encoded as:

```
p(< "a"; 0 > < "L"; 1 >).
```

We implicitly describe net transitions (`Tran` terms) through their incidence matrix (a 3-tuple of `Pbag` terms) and associated tags. A tag includes a `String`, a `Nat` (indicating a priority) and a `Float` (interpreted as a firing rate or a probabilistic weight, depending on whether the priority is zero or greater)..

```
[I,O,H] |-> « S, P, R »
```

When using the associative composition operator `_;_` and the subsort relation `Tran < Net`, it becomes easy to construct nets in a modular way. For example, we can depict the subnet containing transitions ld and ln_0 in Figure 1 (top) as the `Net` term in the listing 1 (the zero-arity operator `nilP` represents an empty multi-set).

Listing 1: A (sub)net

```
[2 . p(< "s" ; 0 >), 1 . p(< "w" ; 0 >) + 1 . p(< "w" ; 1 >), nilP] |-> << "ld", 0, 0.5 >> ;
[1 . p(< "w" ; 0 >), 1 . p(< "a" ; 0 >), 1 . p(< "f" ; 0 > ] |-> << "ln", 0, 0.1 >>
```

A `System` term is the empty juxtaposition (`__`) of a `Net` and a `Pbag` (representing the net's marking). The conditional rewrite rule `firing` specifies the PT firing rule [3], as shown in the listing 2.

Listing 2: PT Firing Rule

```
vars N N' : Net .
vars T : Tran .
var M : Pbag .
crl [firing] : N M => N fire(T, M) if T ; N' := N /\ enabled(T, N M) .
```

[3]Notice the use of a matching equation: The free variables T, N', are matched (:=) against the canonical ground term bound to the variable N.

The predicate `enabled` takes priority into account and relies on `hasConcession`, which determines the 'topological' aspect of the enabling condition:

Listing 3: PT Firing operators

```
vars I O H M : Pbag . var L : Tlab .
op hasConcession : Tran Pbag −> Bool .
eq hasConcession([I,O,H] |−> L, M) = I <= M and−then H > B .
op fire : Tran Pbag −> Pbag .
eq fire([I,O,H] |−> L, M) = M − I + O .
```

A RwSPT is defined by a system module that contains two constant operators, used as aliases:

`op net : -> Net`

`op m0 :  -> Pbag.`

Two equations define their bindings to concrete terms. This module includes `System` rewrite rules R incorporating `firing`. In this paper, we adopt full non-determinism (interleaving semantics): Rewrites take the same priority and have an exponential rate (specified in the rule label but for `firing` rule), so that for the state transition system it holds ($\subseteq$ means subgraph):

$TS(\texttt{net m0}, \{\texttt{firing}\}) \subseteq TS(\texttt{net m0}, R)$.

Transitioning between the `Maude` encoding of PT systems and the PNML format adopted by many PN tools is straightforward and reversible.

4.1 Modularity, symmetries, and lumpability

We have provided net-algebra and net-rewriting operators [8] with a twofold intent: to ease the modeler's task and to enable the construction and modification of large-scale models with nested components by implicitly highlighting their symmetry. A compact *quotient* TS is built using simple manipulation of node labels. This approach outperforms similar ones, including ours integrated into `Maude` [6] and based on traditional graph canonization.

In a context where nets have a mutable structure, identifying behavioral equivalences reduces to a graph *morphism*. PT system morphism must maintain the edges and the marking: In our encoding, a *morphism* between PT systems `(N m)` and `(N' m')` is a bijection ϕ : `places(N)` $\rightarrow$ `places(N')` such that, considering the homomorphic extension of ϕ on multisets, ϕ(`N`) = `N'` and ϕ(`m`) = `m'`. Moreover, ϕ must retain the textual annotations of the place labels and the transition tags. If `N' = N` we speak of *automorphism*, in which case ϕ is a permutation in the set of places.

We refer to a *normal* form that principally involves identifying sets of automorphic (permutable) places: Two markings `m`, `m'` of a net `N` are said automorphic if there is an automorphism ϕ in `N` that maps `m` into `m'`. We denote this `m` $\cong$ `m'`. The equivalence relation $\cong$ is a congruence, that is, it preserves the transition firings and *rates*. The next definition helps us simplify the process.

Definition 4.1 (Symmetric Labeling). *A* `Net` *term is symmetrically labeled if any two maximal sets of places whose labels have the same suffix (possibly empty), which is preceded by pairs with the same tag, are permutable. A* `System` *term is symmetrically labeled if its* `Net` *subterm is.*

In other words, if a `Net` term `N` meets definition 4.1, then for any two maximal subsets of places matching:

$P := \{$`p(L' < w ; i > L)`$\}$, $P' := \{$`p(L'' < w ; j > L)`$\}$,

where `L, L', L'' : Plab, w: String, i, j : Nat` there exists an automorphism (permutation) ϕ such that $\phi(P) = P'$, $\phi(P') = P$, which is extended as an identity to the rest of places[4].

If a `System` term adheres to the previous definition, it can be transformed into a 'normal' form by merely swapping indices on the place labels (e.g., i $\leftrightarrow$ j), while still complying with definition 4.1. This normal form is the most minimal according to a lexicographic order within the automorphism class ($\cong$) implicitly defined by 4.1. However, in contrast to general graph canonization, there is no need for any pruning strategy or backtracking. In simple terms, a monotone procedure is used where the sequence of index swaps does not matter (see [8] for full details). Efficiency is achieved as the normalized form of the subterm of type `Net` is derived through basic "name abstraction", where at each hierarchical level the indices in the structured place labels continuously span from 0 to k, $k \in \mathbb{N}$.

The strategy involves providing a concise set of operators that preserve nets' symmetric labelling. This set includes *compositional* operators (influenced by process algebra) and operators for *manipulating* nets, such as adding/removing components. Rewrite rules require these operators to manipulate `System` terms defined in a modular manner. Additionally, rules must adhere to parametricity conditions (here omitted) that limit the use of non-variable terms in them. We denote this kind of rules as symmetric [8].

Lumpability Under these assumptions, we get a *quotient* TS from a `System` term that retains reachability and meets strong bisimulation.

Let t, t', u, u' be (final) ground terms of sort `System`, and let r be a `System` type rule $r:\ s \Longrightarrow s'$. The notation $t \overset{r(\sigma)}{\Longrightarrow} t'$ means that t is rewritten into t' by r, that is, there exists a ground substitution σ of r's variables such that $\sigma(s) = t$ and $\sigma(s') = t'$.

Property 4.1. *Let t meet Definition 4.1 and r be a symmetric rule.*
If $t \overset{r(\sigma)}{\Longrightarrow} t'$ *then* $\forall u, \phi, t \cong_{\phi} u$*:* $u \overset{r(\phi(\sigma))}{\Longrightarrow} u'$, $t' \cong u'$ *(*u', t' *meet the definition 4.1)*

The TS quotient produced by a term $\hat{t}$ (pre-normalized) is achieved by applying the overloaded operator `normalize` to the right-hand side of the rewriting rules:

```
op normalize : System -> System .
op normalize : Pbag -> Pbag .
```

When a `System` is rewritten using the rule `firing`, only the marking subterm is needed. This implies applying the overloaded operator `normalize` to the subterm `fire(T, M)` in Listing 1.2.

According to property 4.1, because the morphism (index exchange) ϕ preserves the transition rates and we assume that the rules are parameterized, it is feasible to map the TS quotient of $\hat{t}$ onto an isomorphic "lumped" CTMC: In a Markov process's state space, an equivalence relation is considered "strong lumpability" if the cumulative transition rates between any two states within a class to any other class remain consistent. Despite the possibility of establishing a more stringent condition matching strong-bisimulation, that is "exact lumpabability" [5], our attention is focused on the aggregated probability.

Example To demonstrate the aforementioned concepts, we will outline the compositional RwPT model of a distributed production system with graceful degradation (Section 3). Initially, this system is composed of N Production Lines (PL) that share raw materials, with each PL split into K interchangeable branches (listing 4). We start by defining the net transitions. Then we build a Production Line using the `repl&share` operator: The term `PL(K)` represents a Production Line (PL) with `K` symmetric branches, similar to the one shown in Figure 1 (top). The structure of the submodel is expressed by adding a pair

[4]According to the definition of PT morphism, the prefixes `L'` and `L''` are consistent in the textual component.

with the tag "L" to the place labels. For example, `p(< "w" ; 0 > < "L" ; 1 >)` describes the "working" place of robot (line) 1 of the PL. We can also choose to exclude places to share among replicas: In this case, we exclude those representing the "warehouse" (tag "s") and faults (tag "o"). Additionally, we can indicate transitions to share: For instance, "load" and "assembly" are shared.

Listing 4: Modular Specification of a Fault Tolerant Production System

```
fmod FTPL is
 pr NET-OP{SPTlab} .
 ops PL PLA nomPL faultyPL NfaultyPL : NzNat -> Net .
 op faultySys : NzNat -> System .
 op NPL : NzNat NzNat -> Net [memo].
 op NPLsys : NzNat NzNat NzNat -> System .
 ops loadLab asLab failLab workLab : -> Tlab [memo] .
 eq loadLab = << "ld",0, 0.5 >> .
 eq asLab = << "as",0, 2.0 >> .
 eq workLab = << "ln",0, 0.1 >> .
 eq failLab = << "ft",0, 0.001 >> .
 var I : Nat .
 vars N K M : NzNat .
 eq line = [1 . p(< "w" ; 0 >),1 . p(< "a" ; 0 >),1 . p(< "f" ; 0 >) ] |-> workLab .
 eq fault = [1 . p(< "o" ; 0 >) , 1 . p(< "f" ; 0 >), nilP ] |-> failLab .
 eq load = [1 . p(< "s" ; 0 >) , 1 . p(< "w" ; 0 >) , nilP ] |-> loadLab .
 eq ass = [1 . p(< "a" ; 0 >) , 1 . p(< "s" ; 0 >) , nilP ] |-> asLab .
 eq cycle = load ; line ; ass ; fault .
 eq PL(K) = repl&share(cycle, K, "L", p (< "o" ; 0 >) U p(< "s" ; 0 >), asLab U loadLab) .
 eq NPL(N, K) = repl&share(PL(K), N, "PL", p(< "s" ; 0 >), emptyStlab) .
 eq NPLsys(N, K, M) = setMark(setMark(NPL(N, K), "o" "PL", 1), "s", K * M) .
 ...
endfm
```

The term `NPL(N, K)` of type `Net` consists of `N` PLs, each of which contains `K` branches. This net was generated using the `repl&share` operator, which adds the `"PL"` tag to place labels to indicate an additional nesting level. The sharing mechanism ensures each PL gathers `K` raw pieces. The PT net represented by `NPL(2,2)` can be seen in Figure 2, top-right. Furthermore, the term `NPLsys (N, K, M)` of type `System` is a PT system that holds `K*M` tokens in the "warehouse" place, with a single token in each place tagged with `"o"` to trigger fault occurrences within a PL. We can build an identical model using the "symmetric" version of the process algebra ALT operator.

The `System` term generated using the above operators possesses symmetrical labeling (refer to definition 4.1), and its `Net` subterm has already been normalized. Consider, e.g., `NPLsys(2, 2, 1)`. By triggering the conflicting transitions "load", which are initially enabled, the following two markings (essentially, subterms of the `System` terms) can be obtained:

m_1 : `p(< "o"; 0 > < "PL"; 0 >) + p(< "o"; 0 > < "PL"; 1 >) +`
`p(< "w"; 0 > < "L"; 0 > < "PL"; 1 >) + p(< "w"; 0 > < "L"; 1 > < "PL"; 1 >)`

m_2 : `p(< "o"; 0 > < "PL"; 0 >) + p(< "o"; 0 > < "PL"; 1 >) +`
`p(< "w"; 0 > < "L"; 0 > < "PL"; 0 >) + p(< "w"; 0 > < "L"; 1 > < "PL"; 0 >).`

These are automorphic (one can be converted into the other by interchanging `< "PL"; 1 >` $\leftrightarrow$ `< "PL"; 0 >`), but the second marking is the smallest in lexicographic order and hence corresponds to the normalized form.

The following rewrite rule (see the listing 5) encapsulates the self-adjustment of a PL with $K = 2$ in response to a fault, enabling it to function in a diminished capacity (refer to figure 2). This rule deviates slightly from [8], as it is locally activated by a breakdown, leading to a significantly larger TS. Skipping technical details, we point out that the rule meets the parameterity and only employs operators that uphold the definition 4.1, such as `join` , `detach` , `setMark` . Therefore, it retains the symmetrical PT labeling (definition 4.1). The label of the rule contains the exponential rewrite rate as meta-information. There is another rule, not discussed here, that eliminates a faulty and degraded PL from the system.

Listing 5: Rewrite rule of a PL (the label contains the rule's exponential rate)

```
vars S S' S'' : Pbag . vars I J : Nat .
var Sys Sys' : System . var L : Lab .
crl [r1–0.005] : N S => normalize(join(Sys, setMark(setMark(Sys', "w" "fPL", | match(S', "w") |),
    "a" "fPL", | match(S', "a") |)))
if S'' + 1 . p(< "f" ; J > L < "PL" ; I >) := S ∧ N' := nomPL(I) ∧ dead (N' S) ∧ S' := subag(S'', < "PL" ;
    I >) ∧ Sys := detache(N, N') S'' – S' ∧ Sys' := faultySys(notIn(N, "fPL")) .
```

With the model-checking facilities of Maude (in this case, the search command), it is possible to formally demonstrate that for any given N, the quotient transition system has two absorbing states: Every state comprises a deteriorated PL that contains all $2 \cdot M$ materials (unprocessed, except possibly one). This is equivalent to the command below, which yields the same results as its unconditioned counterpart.

```
search NPLsys(N,2,M) =>! F:System such that
net(F:System) == faultyPL /\ B:Pbag := marking(F:System) /\
| match(B:Pbag, "w") | + | match(B:Pbag, "a") | == 2 * M  .
```

5 Obtaining the Lumped CTMC generator from an RwSPT

The CTMC generator entry $Q[i, j]$ is defined as: $\sum_{r\in R} \lambda_r \cdot |S^r_{i,j}|$, where $\lambda_r \in \Re^+$ is a given rate, and $S^r_{i,j} = \{\sigma \mid \hat{t}_i \overset{r(\sigma)}{\Longrightarrow} t_j,\ t_j \cong \hat{t}_j\}$ represents the matches of r resulting in equivalent states. Therefore, to obtain the CTMC infinitesimal generator, it is necessary to quantify instances that correspond to a specific state transition. Our solution uses two operators: the first identifies potential matches for each rule based on the subset of independent variables involved, and the second simulates the rewriting process. These two operators can be "mechanically" defined from the syntax of a rule.

To gain a clearer understanding of the concept, let us examine a simplified scenario that encompasses the vast majority of cases and to which any case can be reduced. We suppose that for every rule $r \in R$:

1. r is "injective", that is, if $t \overset{r(\sigma)}{\Longrightarrow} t' \wedge t \overset{r(\sigma')}{\Longrightarrow} t'$ then $\sigma = \sigma'$,
2. if r is a conditional rule ($r : s \Longrightarrow s'\ if\ cond$) the condition does not contain any rewrite expressions (taking the concrete form $u \Longrightarrow u'$).

Given these assumptions, it is possible to automatically expand a stochastic RwPT specification to produce a quotient TS. The states in this TS encompass all the information required to build the infinitesimal generator of the lumped CTMC, which is isomorphic to the TS.

Listing 6, which is related to the running example, describes a general pattern. To avoid overly technical details of `Maude` syntax, we outline an operator, `rule`, which encodes any rewriting rules except for `firing` (handled separately for efficiency). This operator defines a *partial* mapping where, given a label (defined using a `Tlab`) and a `System` term, it determines the corresponding term-rewriting if feasible: each rewrite rule is tied to an equation. The operator `ruleApp` builds upon `rule`: it computes all potential outcomes of rewriting that term using the rule. It does not execute term normalization. As is typical in `Maude`, the `ruleApp` definition is optimized via tail-recursion. Lastly, `ruleExe`, which extends `ruleApp`, partitions the results of a rule application to a term into "equivalence classes" (sort `Rset`) through normalization: each class is represented by a pair `System <-| Float`, that is the aggregate rate towards a normalized state. The operator `ruleApp` serves as the bulk form of `ruleExe`.

Listing 6: rule encoding for the lumped CTMC

```
vars N N' N'' : Net . vars S S' S'' : Pbag . vars I Imin J : Nat .
vars Sys Sys' : System . var L : Lab . var Sp : Pset . var TL : Tlab .

op rule : Tlab System -> [System] . *** one equation for rule
ceq rule(<< "r1",0, 0.005 >>, N S) = join(Sys, setMark(setMark(Sys', "w" "fPL", | match(S', "w")
    |), "a" "fPL", | match(S', "a") |))
  if S'' + 1 . p(< "f" ; J > L < "PL" ; I >) := S ∧ N' := nomPL(I) ∧ dead (N' S) ∧
    S' := subag(S'', < "PL" ; I >) ∧ Sys := detache(N, N') S'' − S' ∧
    Sys' := faultySys(minNotIn(N, "fPL")) .

ceq rule(<< "r2",0, 0.01 >> , N S) = N'' set(S'' − S', p(< "s" ; 0 >), S[p(< "s" ; 0 >)] + | S' |)
  if S'' + 1 . p(< "f" ; J > L < "fPL" ; I >) := S ∧ N' := faultyPL(I) ∧ dead(N' S) ∧
    N'' := detache(N, N') ∧ N'' =/= emptyN ∧ S' := subag(S'', < "fPL" ; I >) .

*** "rule application" (without normalization)
var SS : Set{System} . var TS : [System] . vars R F : Float .
ops ruleApp : Tlab System -> Set{System} .
eq ruleApp(TL, Sys) = $ruleApp(TL, Sys, emptySS) .
op $ruleApp : Tlab System Set{System} -> Set{System} .
ceq $ruleApp(TL, Sys, SS) = $ruleApp(TL, Sys, SS U TS) if TS := rule(TL, Sys) ∧ TS :: System ∧
    not(TS in SS) .
 eq $ruleApp(TL, Sys, SS) = SS [owise] .
*** "aggregate" rates calculation (with normalization)
op rulexe : Tlab System -> Rset .
eq rulexe(TL, Sys) = $rulexe(rate(TL), ruleApp(TL, Sys), emptyRset) .
op $rulexe : Float Set{System} Rset -> Rset .
eq $rulexe(F, emptySS, RS) = RS .
ceq $rulexe(F, Sys U SS, RS ; Sys' <-| R) = $rulexe(F, SS, RS ; Sys' <-| R + F ) if Sys' :=
    normalize(Sys) .
eq $rulexe(F, Sys U SS, RS) = $rulexe(F, SS, RS ; normalize(Sys) <-| F)
  [owise] .
op allRew : System -> Rset [memo] . *** bulk application
eq allRew(Sys) = rulexe(labr1, Sys) U rulexe(labr2, Sys) .
```

The excerpt in Listing 7 illustrates the augmented state representation which contains detailed information on the (normalized) state transition. The state structure defined by the mixfix constructor `StateTranSys` comprises four fields. The initial pair describes the PT system, while the remaining two fields detail the state transitions caused by the `firing` rule and other rewrites, in that order. As explained, we collect state transitions (rule applications) that share the normalized target for calculating the aggregated rates.

Now, let us examine the `firing` rule (Listing 2): we rephrase it using two related operators, specifically `enabSet`, which determines the set of transitions enabled in a specific marking (or more broadly, the rule's matches), and `fire`, which identifies the resulting markings (the rule's outcomes for all matches), each linked to its respective cumulative rate. The method applied for the `firing` rule can be generalized to any rule, including those that are not injective.

The function `toStateTran` transforms the traditional state representation into a structured format that emphasizes cumulative transition rates. The actual implementation of the firing rule and other transformation rules is simple, as their effects are immediately apparent in the enhanced state information.

Listing 7: TS encoding for the lumped CTMC

```
vars B B' M M' : Pbag . var N : Net . var TS : TagSet . var FS : Fset . var RS : Rset .
var R : Float .
*** description of a system pointing out (aggregate) state−transition rates
op NET:_ M:_ FIRING:_REW:_ : Net Pbag Fset Rset −> StateTranSys [ctor] .
op toStateTran : System −> StateTranSys .
eq toStateTran(N M) = NET: N M: M FIRING: fire(enabSet(N M), M) REW: allRew(N M) .
*** caculates the cumulative firing effect of a net (that is, a set of transitions)
op fire : Net Pbag −> Fset .
*** definition of fire
***
*** implementation of rewrite rules
rl [firing] : NET: N M: B FIRING: (B' <−| R ; FS) REW: RS => toStateTranSPN(N B') .
*** net rewrites
rl [rew] : NET: N M: B FIRING: FS REW: (Sys <−| R ; RS) => toStateTranSPN(Sys) .
```

When considering `toStateTran(NPLsys(2,2,2))`, which aligns with the PT net at the top of Figure 2, the resulting quotient TS comprises 295 states compared to the 779 states in the standard TS. The quotient graph's state transitions often correspond to multiple matches. For instance, the initial state (the term above) includes two 'load' instances and four 'fault' instances that lead to markings with identical normal forms. Consequently, the combined rates are $2 \cdot 0.5$ and $4 \cdot 0.001$. Equivalent rewrites of the net structure are observed when $N > 2$.

5.1 Experimental Evidence

We conclude by showcasing the experimental validation of the method alongside a straightforward demonstration for calculating standard performance metrics. The results of the final-state location command are shown in Table 1 (above). This was carried out using Linux WSL on an 11th-gen Intel Core i5 with 40GB RAM. The state spaces align with those of the corresponding lumped CTMC. It is evident that analysis of large models is achievable by leveraging the model's symmetry. Note that the number of absorbing states in the TS quotient remains unchanged with N. Even though a redundant state representation was used to construct the lumped CTMC directly, the efficiency of the `Maude` rewriting engine allowed us to estimate a time overhead of no more than 80%.

Table 1: Ordinary vs Quotient TS of `NPLsys(N,2,2)` † `search` timed out after 10 h

	Ordinary		Quotient	
`N`	states(final)	time (sec)	states(final)	time (sec)
1	60(2)	0	42(2)	0
2	779(4)	0.1	295(2)	0.1
3	6101(6)	4.8	1059(2)	0.9
4	37934(8)	69	2764(2)	3.6
5	204362(10)	818	5970(2)	10
6	1000187(12)	13930	11367(2)	27
7	-	†	19775(2)	65
8	-	†	32144(2)	186
9	-	†	49554(2)	569
10	-	†	73215(2)	2450

According to [8], the performance of modular RwPT was evaluated against symmetric nets (SN, previously referred to as well-formed nets) [9], which are colored Petri nets that produce a symbolic reachability graph (SRG) comparable (in its stochastic extension) to a lumped CTMC. As N and K values rise, the state aggregation level in modular RwPT drastically surpasses that of SN. For example, with N=10, K=3, and M=3, the state aggregation level is around 45 times greater than SN, and with N=10, K=4, and M=3, it is approximately 200 times greater than SN. You can replicate the experiments following the guidelines at https://github.com/lgcapra/rewpt/tree/main/modSPT/readme.

Figure 3 shows the system throughput, while 4 shows its reliability as a time function. As expected, both metrics decrease with time; additionally, the scenario that involves more replicas demonstrates increased throughput and enhanced reliability. To evaluate the system's performance, Figure 5 shows the throughput while the system is operational, which is the ratio between the graphs in Figures 3 and 4. It can be seen that the throughput is close to that of a single line, which, given the parameters, is $1/202.5 = 4.98E-03$. The inflection point at around time 800 in both curves represents the system's reconfiguration time. The increased execution time of the job is a result of a system failure.

The overall trend is also noticeable when we look at larger values of `N`. As `N` increases, both reliability and throughput curves show significant improvements. However, we observe an asymptotic trend when `N` is greater than 6. Our interpretation is that beyond a certain point, the benefit of using a higher number of replicas is outweighed by the higher fault rate and the increased configuration overhead.

6 Conclusion and Future Work

We have created a Lumped Markov process for modular, rewritable stochastic Petri nets (RwPT), which serves as a robust model for analyzing adaptive distributed systems encoded in `Maude`. RwPT models, assembled and manipulated through a compact set of (algebraic) operators, display structural symmetries leading to an efficient quotient state transition graph. By providing an example of a gracefully degrading system, we have demonstrated a semi-automatic method for deriving the CTMC infinitesimal generator from the RwPT quotient graph. Future work will, on one hand, delve into exploring orthogonal structured solutions and, on the other, focus on fully implementing the process and integrating it with graphical tools such as `DrawNET` (https://www.draw-net.com/). At the same time, we aim to expand the approach: firstly, to derive a lumped Markov process from rewritable GSPN, and secondly, to extract the infinitesimal CTMC generator from any `Maude` system module.

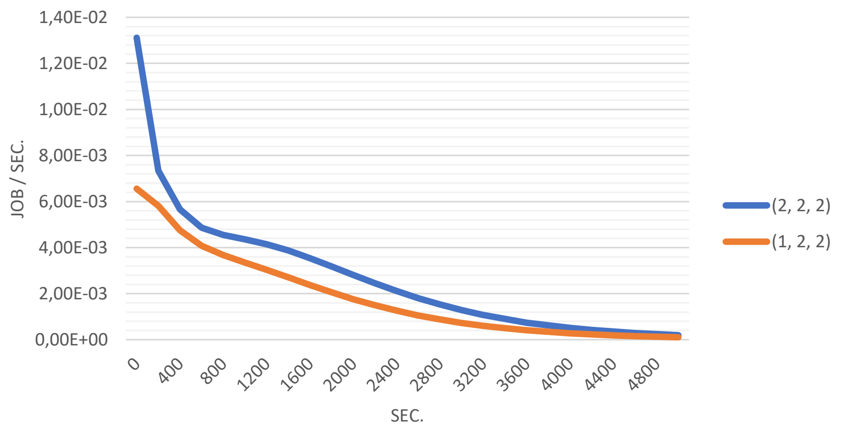


Figure 3: System Throughput.

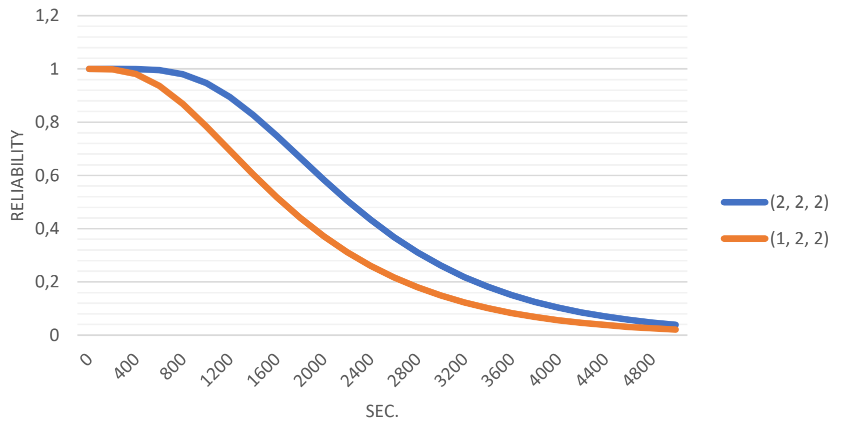


Figure 4: System Reliability.

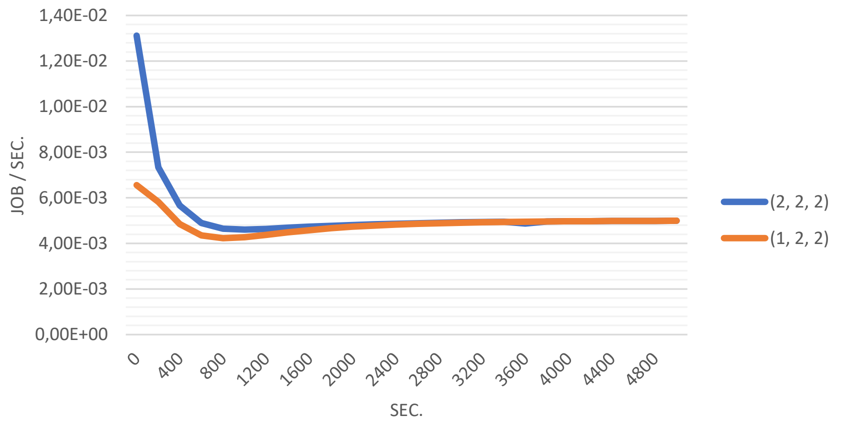


Figure 5: System Throughput conditioned to its reliability.

References

[1] László Babai (2016): *Graph Isomorphism in Quasipolynomial Time [Extended Abstract]*. In: *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, Association for Computing Machinery, New York, NY, USA, p. 684–697, doi:10.1145/2897518.2897542.

[2] Paulo E. S. Barbosa, João Paulo Barros, Franklin Ramalho, Luís Gomes, Jorge Figueiredo, Filipe Moutinho, Anikó Costa & André Aranha (2011): *SysVeritas: A Framework for Verifying IOPT Nets and Execution Semantics within Embedded Systems Design*. In Luis M. Camarinha-Matos, editor: *Technological Innovation for Sustainability - Second IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2011, Costa de Caparica, Portugal, February 21-23, 2011. Proceedings*, *IFIP Advances in Information and Communication Technology* 349, Springer, pp. 256–265, doi:10.1007/978-3-642-19170-1_28.

[3] Adel Bouhoula, Jean-Pierre Jouannaud & José Meseguer (2000): *Specification and proof in membership equational logic*. *Theoretical Computer Science* 236(1), pp. 35–132, doi:10.1016/S0304-3975(99)00206-6.

[4] Roberto Bruni & José Meseguer (2003): *Generalized Rewrite Theories*. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow & Gerhard J. Woeginger, editors: *Automata, Languages and Programming*, Springer-Verlag, Berlin, Heidelberg, pp. 252–266, doi:10.1007/3-540-45061-0_22.

[5] Peter Buchholz (1994): *Exact and Ordinary Lumpability in Finite Markov Chains*. *Journal of Applied Probability* 31(1), pp. 59–75, doi:10.2307/3215235.

[6] Lorenzo Capra (2022): *Canonization of Reconfigurable PT Nets in Maude*. In Anthony W. Lin, Georg Zetzsche & Igor Potapov, editors: *Reachability Problems*, Springer International Publishing, Cham, pp. 160–177, doi:10.1007/978-3-031-19135-0_11.

[7] Lorenzo Capra (2022): *Rewriting Logic and Petri Nets: A Natural Model for Reconfigurable Distributed Systems*. In Raju Bapi, Sandeep Kulkarni, Swarup Mohalik & Sathya Peri, editors: *Distributed Computing and Intelligent Technology*, Springer International Pub., Cham, pp. 140–156, doi:10.1007/978-3-030-94876-4_9.

[8] Lorenzo Capra & Michael Köhler-Bußmeier (2024): *Modular rewritable Petri nets: An efficient model for dynamic distributed systems*. *Theoretical Computer Science* 990, p. 114397, doi:10.1016/j.tcs.2024.114397.

[9] G. Chiola, C. Dutheillet, G. Franceschinis & S. Haddad (1997): *A symbolic reachability graph for coloured petri nets*. *Theoretical Computer Science* 176(1), pp. 39 – 65, doi:10.1016/S0304-3975(96)00010-2.

[10] Giovanni Chiola, Marco Ajmone Marsan, Gianfranco Balbo & Gianni Conte (1993): *Generalized Stochastic Petri Nets: A Definition at the Net Level and Its Implications*. *IEEE Trans. Software Eng.* 19, pp. 89–107, doi:10.1109/32.214828.

[11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso M. Oliet, Jos'e Meseguer & Carolyn Talcott (2007): *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Springer, doi:10.1007/978-3-540-71999-1.

[12] Julia Padberg & Alexander Schulz (2016): *Model Checking Reconfigurable Petri Nets with Maude*. In Rachid Echahed & Mark Minas, editors: *Graph Transformation*, Springer International Publishing, Cham, pp. 54–70, doi:10.1007/978-3-319-40530-8_4.

[13] W. Reisig (1985): *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, doi:10.1007/978-3-642-69968-9.

Tracer: A Tool for Race Detection in Software Defined Network Models

Georgiana Caltais
University of Twente
The Netherlands
g.g.c.caltais@utwente.nl

Mahboobeh Zangiabady
University of Twente
The Netherlands
m.zangiabady@utwente.nl

Ervin Zvirbulis
University of Twente
The Netherlands
e.zvirbulis@student.utwente.nl

Software Defined Networking (SDN) has become a new paradigm in computer networking, introducing a decoupled architecture that separates the network into the data plane and the control plane. The control plane acts as the centralized brain, managing configuration updates and network management tasks, while the data plane handles traffic based on the configurations provided by the control plane. Given its asynchronous distributed nature, SDN can experience data races due to message passing between the control and data planes. This paper presents Tracer, a tool designed to automatically detect and explain the occurrence of data races in DyNetKAT SDN models. DyNetKAT is a formal framework for modeling and analyzing SDN behaviors, with robust operational semantics and a complete axiomatization implemented in Maude. Built on NetKAT, a language leveraging Kleene Algebra with Tests to express data plane forwarding behavior, DyNetKAT extends these capabilities by adding primitives for communication between the control and data planes. Tracer exploits the DyNetKAT axiomatization and enables race detection in SDNs based on Lamport vector clocks. Tracer is a publicly available tool.

1 Introduction

Traditional network devices have been called "the last bastion of mainframe computing" [7]. Since the 1970s, network design principles have remained fundamentally unchanged, maintaining their core structure for nearly four decades. One of such fundamentals is the handling of the data and control planes. Intuitively, the data plane is a distinct functional layer in networking responsible for the forwarding of data packets between network devices. The control plane is another layer responsible for network control including policy enforcing and routing configuration. In a traditional network, each switch autonomously manages its interpretation of the control plane as illustrated in Figure 1. This architectural rigidity increases complexity in network maintainability due to the necessity of configuring each switch individually.

In response, the concept of software-defined networking (SDN) has emerged. The main difference is the separation of the data and control planes and consolidation of the management over the control plane in a centralized location as illustrated in Figure 2. SDN architectures comprise central controllers and programmable switches that communicate via standardized protocols. The former respond to network events such as new connections from hosts, topology changes, and shifts in traffic load, by reprogramming the switches accordingly (as indicated by the orange dotted arrows in the figure). Such an approach enhances network controllability and adaptability in real-time scenarios. SDN is being adopted across various leading tech companies and cloud service providers to enhance the agility, efficiency, and scalability of their data center networks. For instance, Google's B4 that connects Google's data centers across the world uses a centralised SDN controller that manages the entire network. Microsoft has also

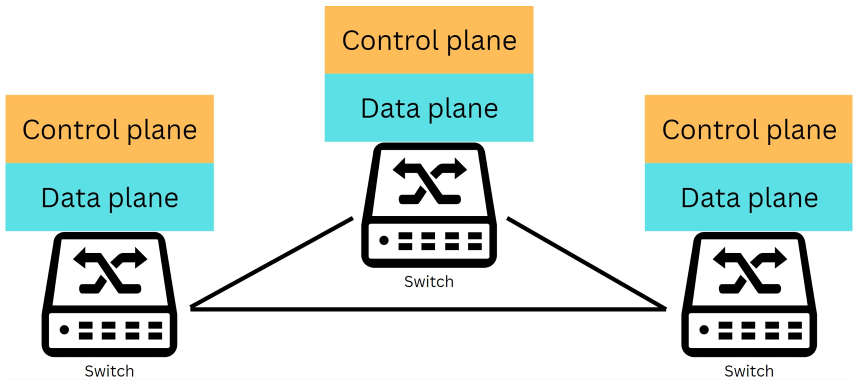


Figure 1: Traditional network setup

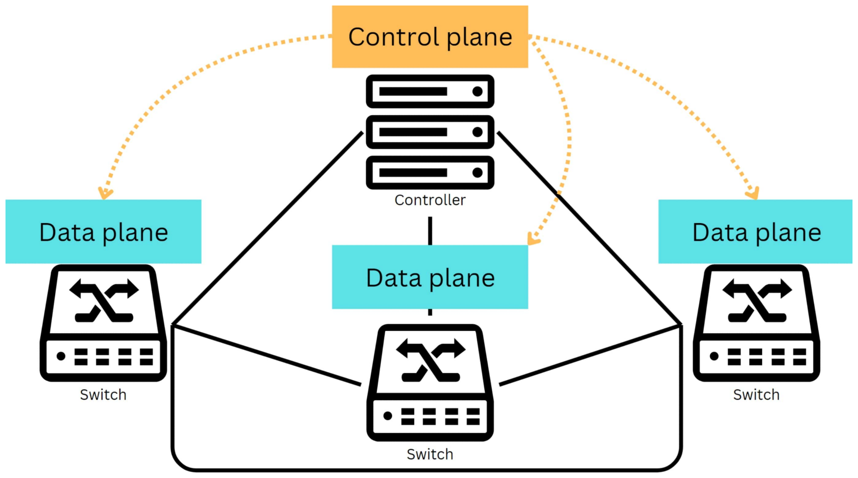


Figure 2: Software-defined network setup

implemented SDN extensively in its Azure cloud infrastructure. Amazon employs SDN to support its scalable and flexible cloud services.

SDNs are highly concurrent systems, designed to handle numerous simultaneous operations. Consequently, they are prone to data races. The latter can lead to undesired outcomes/behaviours of the SDN, especially if the races correspond to concurrency between the data and the control planes.

Let us consider Figure 3 (inspired from [4]), illustrating a basic example of an SDN consisting of: (i) a switch with two ports (1 and 2), (ii) one controller communicating with the switch, and (iii) two hosts (Host 1 and Host 2) that can send/receive packets to/from the switch via the aforementioned ports. Assume the following over-simplified scenario: The switch is configured to allow any traffic from port 1 to port 2. When the switch encounters a "blocking" flag, it notifies the controller and continues forwarding subsequent packets until a new forwarding policy ("drop everything", in this case) is received from the controller. If Host 1 sends a packet flagged "blocking" to the switch, a data race may occur. The race arises because the outcome for a new packet depends on the timing. The new packet will either be forwarded according to the existing forwarding policy installed in the switch, if it arrives before the blocking rule from the controller, or it will be dropped if the blocking rule is in place first.

In this paper, we propose Tracer [1], a tool for the automated detection of data races in SDNs. Tracer builds around DyNetKAT [5], a formal framework for the rigorous modelling and analysis of SDNs.

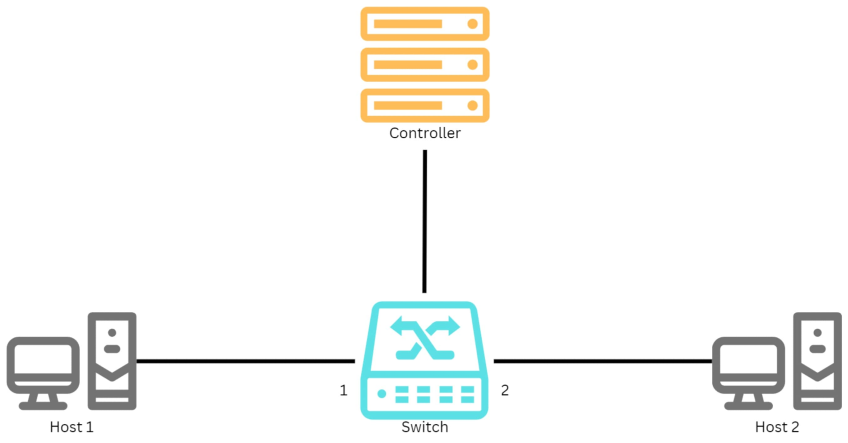


Figure 3: Running example inspired from [10]

DyNetKAT can encode and simulate packet forwarding within SDNs, together with the actual communication between the control and data planes (or, dynamic network reconfigurations). The DyNetKAT language is supported by a rigorous operational semantics and a sound and complete axiomatisation enabling reasoning about equivalence of DyNetKAT programs, and associated packet forwarding behaviour in a fully automated fashion. Intuitively, Tracer takes as input DyNetKAT models and checks whether data races between the control and data planes occur, by following the symbolic approach in [4]. Furthermore, Tracer provides explanations of how such races can be enabled via (minimal) sequences of packets fed to the network. Such explanations can serve as a great debugging aid for the network administrators.

As mentioned in [4], several methods have been developed to detect race conditions in SDNs, including ConGuard [13] and SDNRacer [6]. These tools identify race condition vulnerabilities by analyzing dynamically generated log files to construct an event graph where happens-before edges connect events, and race conditions manifest as partially ordered events. The Spin model checker, as discussed in [12], has also been employed to detect race conditions through runtime monitoring of events in SDNs. In contrast, this paper is based on a *static approach to identify races* in SDNs, eliminating the need for dynamic log generation from a network.

Our contributions: As previously mentioned, in this paper we introduce Tracer [1], a tool for the automated detection of races in SDN models encoded in DyNetKAT, based on the theoretical framework in [4]. In short, Tracer exploits the symbolic semantics of DyNetKAT in [4] and uses Lamport vector clocks for detecting races entailed by the concurrent message passing between the SDN control and data planes as in [4]. Furthermore, Tracer provides explanations of how such races are enabled by computing minimal sets of network packets that lead to not well-behaved communication scenarios. The instructions for installing and running Tracer are publicly available in [1].

Structure of paper: In Section 2, we briefly recall (Dy)NetKAT and introduce our running example. In Section 3, we present the idea behind vector clocks for race detection in distributed systems. The symbolic semantics of DyNetKAT enriched with vector clocks is recalled in Section 4. Our tool, Tracer, is introduced in Section 5. We draw the conclusions and provide pointers to future work in Section 6.

2 Overview of DyNetKAT

DyNetKAT [5] serves as a framework for representing and analyzing the behaviors of SDNs, such as packet forwarding and the interaction between the control and data planes. DyNetKAT is an extension of NetKAT [2], which is a language based on Kleene Algebra with Tests [8], tailored for modeling and analyzing data plane forwarding. DyNetKAT introduces concurrency to NetKAT, in order to support dynamic reconfigurations of the data plane, such as the installation of new forwarding rules, in line with control plane protocols.

Figure 4 illustrates the syntax and semantics of the NetKAT language. Network packets are encoded in (Dy)NetKAT as collections of fields, and associated values ranging over finite domains: $\{f_1 = v_1, \ldots, f_n = v_n\}$. For instance, a packet σ of type SSH residing at port 1 of switch SW_A, with destination $Host_1$, can be conveniently denoted as $\sigma \triangleq \{type = SSH, pt = 1, SW = SW_A, dst = Host_1\}$. The main syntactic elements of NetKAT include primitives (Pr) for dropping incoming packets $(\mathbf{0})$ and accepting incoming packets without further processing $(\mathbf{1})$. NetKAT primitives can filter out packets based on tests $(f = n)$ and their disjunction $(+)$, conjunction $(\cdot)$ and negation $(\neg)$. NetKAT policies (N) can also be used for packet fields modifications $(f \leftarrow n)$, or to express packet multicasting $(+)$, composition of policies $(\cdot)$ and iteration $(^*)$. The operator **dup** is designed for building histories of packets processed by an SDN dataplane encoded in NetKAT. The denotational semantics of NetKAT is defined over sets of packet histories as in Figure 4; an intuitive description of its operators has been provided earlier. Furthermore, NetKAT has a sound and complete axiomatization that has been effectively used to reason about packet reachability within NetKAT models.

NetKAT Syntax:

$$\begin{array}{rcl} Pr & ::= & \mathbf{0} \mid \mathbf{1} \mid f = n \mid Pr + Pr \mid Pr \cdot Pr \mid \neg Pr \\ N & ::= & Pr \mid f \leftarrow n \mid N + N \mid N \cdot N \mid N^* \mid \mathbf{dup} \end{array}$$

NetKAT Semantics:

$$\begin{array}{rcl} [\![\mathbf{1}]\!](h) & \triangleq & \{h\} \\ [\![\mathbf{0}]\!](h) & \triangleq & \{\} \\ [\![f = n]\!]\,(\sigma{::}h) & \triangleq & \begin{cases} \{\sigma{::}h\} & \text{if } \sigma(f) = n \\ \{\} & \text{otherwise} \end{cases} \\ [\![\neg a]\!]\,(h) & \triangleq & \{h\} \setminus [\![a]\!]\,(h) \\ [\![f \leftarrow n]\!]\,(\sigma{::}h) & \triangleq & \{\sigma[f := n]{::}h\} \\ [\![p + q]\!]\,(h) & \triangleq & [\![p]\!]\,(h) \cup [\![q]\!]\,(h) \end{array} \qquad \begin{array}{rcl} [\![p \cdot q]\!]\,(h) & \triangleq & ([\![p]\!] \bullet [\![q]\!])\,(h) \\ [\![p^*]\!]\,(h) & \triangleq & \bigcup_{i \in N} F^i\,(h) \\ F^0\,(h) & \triangleq & \{h\} \\ F^{i+1}\,(h) & \triangleq & ([\![p]\!] \bullet F^i)\,(h) \\ (f \bullet g)(x) & \triangleq & \bigcup \{g(y) \mid y \in f(x)\} \\ [\![\mathbf{dup}]\!]\,(\sigma{::}h) & \triangleq & \{\sigma{::}(\sigma{::}h)\} \end{array}$$

Figure 4: NetKAT: Syntax and Semantics [2]

The syntax of DyNetKAT is defined on top of the **dup**-free fragment of NetKAT as in (1). The constant $\perp$ denotes a DyNetKAT process without behaviour. Sequential composition of DyNetKAT policies D is denoted by ; . The operator $||$ encodes concurrent behaviours of DyNetKAT policies, whereas $\oplus$ stands for non-deterministic choice. (A)synchronous communication in DyNetKAT is modeled in an ACP [3]-style via message sending operators $x!N;D$ and receiving operators $x?N;D$. Intuitively, messages N (e.g., NetKAT flow tables) can be exchanged via channels x as a result of the communication between the control and data planes. As soon as such a new forwarding policy N is received via x, the continuation D can update its behaviour according to N. This would correspond to installing a new forwarding policy N in the dataplane. Variables X enable defining recursive DyNetKAT policies.

$(\mathbf{cpol}^{\checkmark}_{\_;})\dfrac{\sigma' \in [\![p]\!](\sigma::\langle\rangle)}{(p;q,\sigma :: H,H') \xrightarrow{(\sigma,\sigma')} (q,H,\sigma' :: H')}$	$(\mathbf{cpol_X})\dfrac{(p,H_0,H_1) \xrightarrow{\gamma} (p',H_0',H_1')}{(X,H_0,H_1) \xrightarrow{\gamma} (p',H_0',H_1')} X \triangleq p$
$(\mathbf{cpol}_{\_\oplus})\dfrac{(p,H_0,H_0') \xrightarrow{\gamma} (p',H_1,H_1')}{(p \oplus q,H_0,H_0') \xrightarrow{\gamma} (p',H_1,H_1')}$	$(\mathbf{cpol}_{\_\Vert})\dfrac{(p,H_0,H_0') \xrightarrow{\gamma} (p',H_1,H_1')}{(p\Vert q,H_0,H_0') \xrightarrow{\gamma} (p'\Vert q,H_1,H_1')}$
$(\mathbf{cpol}_{\bullet})\dfrac{}{(x \bullet p;q,H,H') \xrightarrow{x \bullet p} (q,H,H')} \quad \bullet \in \{?,!\}$	
$(\mathbf{cpol}_{\clubsuit\spadesuit})\dfrac{(q,H,H') \xrightarrow{x\clubsuit p} (q',H,H') \quad (s,H,H') \xrightarrow{x\spadesuit p} (s',H,H')}{(q\Vert s,H,H') \xrightarrow{\mathbf{rcfg(x,p)}} (q'\Vert s',H,H')} \quad \clubsuit = ?\ \spadesuit = !$ or $\clubsuit = !\ \spadesuit = ?$	

$$\gamma ::= (\sigma,\sigma') \mid x!q \mid x?q \mid \mathbf{rcfg}(\mathbf{x},\mathbf{q})$$

Figure 5: DyNetKAT: Operational Semantics (relevant excerpt)

$$\begin{array}{lll} N & ::= & \text{NetKAT}^{-\mathbf{dup}} \\ D & ::= & \bot \mid N;D \mid x?N;D \mid x!N;D \mid D\Vert D \mid D\oplus D \mid X \\ & & X \triangleq D \end{array} \tag{1}$$

The operational semantics of DyNetKAT is given in Fig. 5, over tuples of shape (D,H,H'), where D is a DyNetKAT policy, H is the list of packets waiting to be processed by the network, and H' is the history of packets being processed according to the forwarding rules in the data plane. Rule $(\mathbf{cpol}^{\checkmark}_{\_;})$ in Fig. 5, for instance, processes the current packet σ (at the top of the waiting list) according to the NetKAT flow table encoded by p. The possibly modified packet is σ', and a corresponding transition $\xrightarrow{(\sigma,\sigma')}$ can be observed in the behaviour Labelled Transition System (LTS) of the DyNetKAT model. σ' is added to the history H', and the execution of the model proceeds with the continuation q and the remaining waiting packets in H. Rule $(\mathbf{cpol}_{\clubsuit\spadesuit})$, for instance, encodes synchronous communication in DyNetKAT: a new forwarding rule or NetKAT policy p is communicated via channel x in a handshake between two parallel SDN components $q\Vert s$ (e.g., one controller q and one switch s). The handshake entails an execution $\xrightarrow{\mathbf{rcfg(x,p)}}$ within the DyNetKAT model. Rules $(\mathbf{cpol}_{\_\oplus})$ and $(\mathbf{cpol}_{\_\Vert})$ and their symmetric counterparts define non-deterministic choice and parallel composition, respectively, in a standard fashion. Rule $(\mathbf{cpol_X})$ simply replaces recursive variables with their definitions. Rule $(\mathbf{cpol}_{\bullet})$ encodes the axioms for asynchronous communication. Furthermore, DyNetKAT has an ACP-like sound and complete axiomatisation for LTS bisimilarity. A complete and thorough presentation of the DyNetKAT formal framework can be found in [5].

2.1 Running Example

Next, we illustrate the DyNetKAT framework by means of an example. Consider the scenario in Figure 3. A possible encoding in DyNetKAT is given in (2), as follows.

$$\begin{array}{rcl} SW & \triangleq & (flag = regular) \cdot (pt = 1) \cdot (pt \leftarrow 2); SW \oplus \\ & & (flag = blocking) \cdot (pt = 1); ((Help \,!\, 1); SW) \oplus \\ & & (Up \,?\, 1); SW' \\ SW' & \triangleq & \mathbf{0}; \bot \\ C & \triangleq & (Help \,?\, 1); ((Up \,!\, 1); C) \end{array} \tag{2}$$

We write $(flag = regular)$ for packets **not** of the blocking type. Whenever such a packet arrives at port 1 $(pt = 1)$ of the switch (SW), it gets forwarded to port 2 $(pt \leftarrow 2)$. Then, the switch continues recursively (denoted by $;SW$). Alternatively (denoted by $\oplus$), we write $(flag = blocking)$ to encode matching of packets of blocking type. Whenever such a packet arrives at port 1, the switch informs the controller that a new forwarding rule needs to be installed (denoted by sending the message $Help!1$). The new blocking behaviour is announced to the switch via $Up!1$. Upon receiving the message $Up?1$, the forwarding table of SW is updated to SW'. The latter drops any incoming packet $(\mathbf{0})$ and irreversibly stops from processing packets $(\bot)$. The controller (C) repeatedly listens on channel $Help?$ for requests from the switch, and instructs the switch to install the blocking behaviour SW' via $Up!1$.

3 Vector Clocks

SDN is a paradigm that falls under the definition of a distributed system [11]. In this case, the components are controllers and switches, and the whole network represents a distributed system. In distributed systems like SDNs, a data race means that switches and controllers perform actions concurrently, possibly leading to undesired behaviours. As illustrated in the introduction, for instance, there might be the case that due to concurrency, the network still forwards unsafe packets in between a new forwarding policy request, and the actual installation of the new forwarding rules. We call these *data races between the control and data planes*. One possible approach to detecting data races is the use of vector clocks [9]. Each such clock is associated with a component in the distributed system, and it consists of a vector of size equal with the number of components in the system. Each vector entry in a clock counts actions performed by a distinct component. (In the context of DyNetKAT models, for instance, actions stand for packet forwarding or reconfigurations between the control and data planes.) Each component in the system has its own copy of a vector clock as illustrated in Figure 6.

As shown in Figure 6, in step 0 all three clocks are initialized with zeros. When a component performs an individual action (i.e., no message-passing involved), it increments its own index in its local copy of the vector clock. The rest of the entries in the clock, as well as the clocks of the other components, are unchanged (see, e.g., step ① or ⑤ of component A). Both steps ② and ③ correspond to a similar scenario, but within component C.

Synchronous communication is handled as follows: Once a component sends a message, it first increments its clock, and then sends it along with the message, creating a timestamped message (step ④.1). Upon its arrival, the receiver updates the rest of the clock entries in the local copy if the corresponding entries in the message timestamp are greater, and then it increments its clock entry (step ④.2). Note that step 4 consists of two parts capturing the synchronous sending and receiving of a message in one time frame (i.e., caputers a handshake communication).

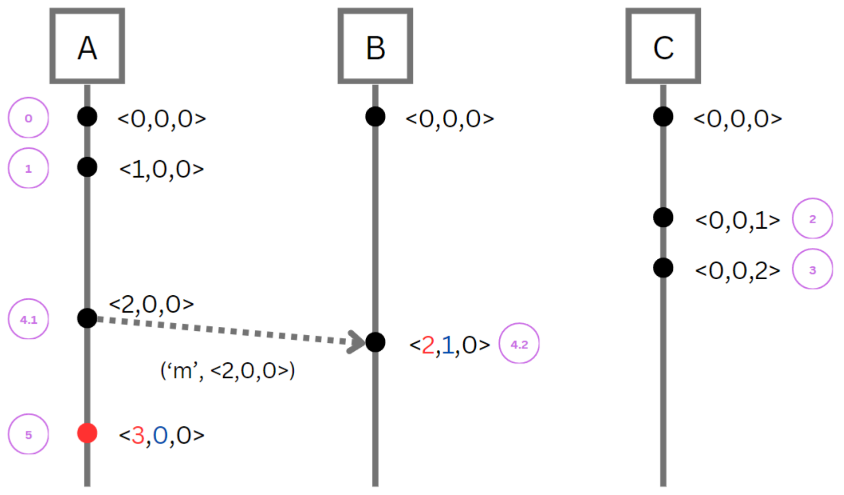


Figure 6: Vector clocks in a system with three parallel components A, B and C

To understand how vector clocks help detect data races we first need to know what does it mean for two vector clocks to be comparable. Consider two vector clocks V_i and V_j of size k. The clocks are comparable if:

$$V_i[x] \leq V_j[x] \qquad \forall x \in \{1,\dots,k\}$$

or

$$V_i[x] \geq V_j[x] \qquad \forall x \in \{1,\dots,k\}$$

Such comparable pair of vector clocks indicates that the associated components did not run concurrently. If there exist $x, y \in \{1,\dots,k\}$ with $x \neq y$ such that:

$$(V_i[x] \not\leq V_j[x] \quad \wedge \quad V_i[y] \not\geq V_j[y])$$

then we conclude that components associated with V_i and V_j operate concurrently, implying a data race. Steps ⑤ and ④.2 for instance, witness concurrent behaviour between A and B. Similarly for ⑤ and ②, etc.

4 Overview of Symbolic DyNetKAT

In this section, we briefly recall the contribution in [4] that introduces a symbolic operational semantics of DyNetKAT, enriched with vector clocks for detecting races between the control and data planes. The most important idea behind the symbolic DyNetKAT reduces to exploiting the so-called DyNetKAT *head normal forms* that enable simulating packet processing within SDN models in a purely syntactic fashion, without actual packets being "fed" to the network.

The idea is as follows: each (guarded) DyNetKAT policy d can be equivalently expressed (based on its complete axiomatisation in [5]) as a sum ($\oplus$) of DyNetKAT policies of shape $\alpha \cdot \pi ; d'$ or $\mathbf{rcfg}(\mathbf{x},\mathbf{n}) ; d'$. Here, α stands for a so-called *complete test* $(f_1 = v_1) \cdot \ldots \cdot (f_n = v_n)$ encoding all the conditions an incoming packet has to match within a flow table, in order to be forwarded accordingly. Each packet passing a complete test as before is, in fact, a packet of shape $\sigma_\alpha \triangleq \{f_1 = v_1, \dots, f_n = v_n\}$; so, a complete test encodes an incoming packet. A *complete assignment* π as before, is a policy $(f_1 \leftarrow$

$v'_1) \cdot \ldots \cdot (f_n \leftarrow v'_n)$ encoding how the packet matching the complete test is processed by the data plane. Basically, a complete assignment encodes a forwarded/processed packet $\sigma_\pi \triangleq \{f_1 = v'_1, \ldots, f_n = v'_n\}$. It is, therefore, easy to understand that the symbolic semantics of DyNetKAT can be defined based on such normal forms which entail transitions of shape $\xrightarrow{(\sigma_\alpha,\sigma_\pi)}$ and $\xrightarrow{\mathbf{rcfg(x,n)}}$, respectively, without the need of actual packets.

In [4], each SDN encoding a set of parallel switches (S_i) and controllers (C_j)

$$S_1 \,||\, \ldots \,||\, S_n \,||\, C_1 \,||\, \ldots \,||\, C_m \tag{3}$$

is enriched with vector clocks $\vec{c}_k$ associated with each component

$$S_{1\vec{c}_1} \,||\, \ldots \,||\, C_{m\vec{c}_m} \tag{4}$$

entailing DyNetKAT symbolic operational rules. For instance:

$$(\mathbf{Symb}_{\checkmark})\frac{p_i \in \text{NetKAT}^{-\mathbf{dup}} \quad n.f.(p_i) = \Sigma_{\alpha_i\cdot\pi_i\in\mathscr{A}}\alpha_i\cdot\pi_i}{(p_i;q_i)_{\vec{c}_i} \,||\, \Pi_{\substack{1\le j\le n\\ j\neq i}} d_{j\vec{c}_j} \xrightarrow{(\sigma_{\alpha_i},\sigma_{\pi_i})} (q_i)_{\vec{c}_i[i]++} \,||\, \Pi_{\substack{1\le j\le n\\ j\neq i}} d_{j\vec{c}_j}}$$

is the symbolic counterpart of ($\mathbf{cpol}^{\checkmark}_{\_;}$), where the vector clock $\vec{c}_i$ of the "evolving" component $p_i\,;q_i$ is incremented in accordance with the semantics of the vector clocks in Section 3, and the input packet σ_{α_i} and the processed packet σ_{π_i} defining this step $\xrightarrow{(\sigma_{\alpha_i},\sigma_{\pi_i})}$ are entailed based on the normal form of p_i (note that normal forms exist for NetKAT as well [2]).

Here we write:

$$P_{i\vec{c}_i} \,||\, \Pi_{\substack{1\le j\le k\\ j\neq i}} P_{j\vec{c}_j}$$

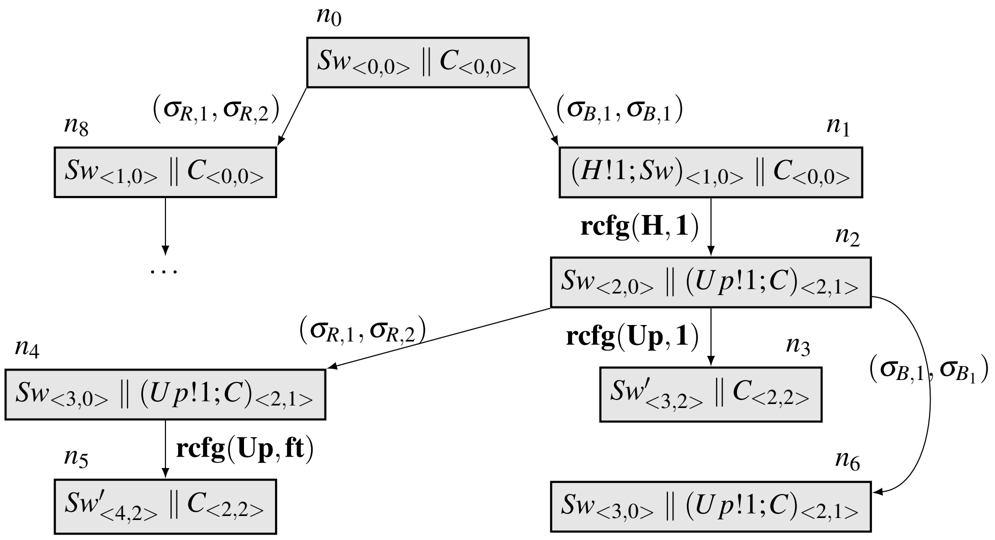
to denote

$$P_{1\vec{c}_1} \,||\, \ldots \,||\, P_{k\vec{c}_k}$$

The symbolic rule for handshake (i.e., the counterpart of ($\mathbf{cpol}_{\_||}$)) is defined is a similar fashion, where both vector clocks of the communicating SDN components are updated, and the transition step is marked as $\xrightarrow{\mathbf{rcfg(x,p)}}$:

$$(\mathbf{Symb}_{||})\frac{h.n.f(q_i) \triangleq x!q;d_i \oplus r_i \quad h.n.f(q_k) \triangleq x?q;d_k \oplus r_k}{\begin{array}{c}(q_1)_{\vec{c}_1} \,||\, \ldots \,||(q_i)_{\vec{c}_i} \,||\, \ldots \,||(q_k)_{\vec{c}_k} \,||\, \ldots \,||(q_n)_{\vec{c}_n} \\ \xrightarrow{\mathbf{rcfg(x,q)}} \\ (q_1)_{\vec{c}_1} \,||\, \ldots \,||(d_i)_{\vec{c}_i[i]++} \,||\, \ldots \,||(d_k)_{(max(\vec{c}_i[i]++,\vec{c}_k))[k]++} \,||\, \ldots \,||(q_n)_{\vec{c}_n}\end{array}}$$

Figure 7 illustrates the symbolic execution of the SDN in (2). For brevity of notation, we write: $\sigma_{B,1}$ to denote a packet $\{flag = blocking, pt = 1\}$, $\sigma_{R,1}$ to encode a packet $\{flag = regular, pt = 1\}$ and $\sigma_{R,2}$ in lieu of $\{flag = regular, pt = 2\}$. As intuitively explained in Section 1: if Host 1 starts sending blocking traffic to the switch on port 1, a data race may occur. The race arises because the outcome for a new packet depends on the timing. The new packet will either be (i) forwarded according to the existing forwarding policy installed in the switch, if it arrives before the blocking rule from the controller, or (ii) it will be dropped if the blocking rule is in place first. Case (i) matches the symbolic execution



n_0
$Sw_{<0,0>} \parallel C_{<0,0>}$
$(\sigma_{R,1}, \sigma_{R,2})$
$(\sigma_{B,1}, \sigma_{B,1})$
n_8
$Sw_{<1,0>} \parallel C_{<0,0>}$
n_1
$(H!1;Sw)_{<1,0>} \parallel C_{<0,0>}$
$\mathbf{rcfg}(\mathbf{H},\mathbf{1})$
n_2
$Sw_{<2,0>} \parallel (Up!1;C)_{<2,1>}$
$(\sigma_{R,1}, \sigma_{R,2})$
$\mathbf{rcfg}(\mathbf{Up},\mathbf{1})$
n_4
$Sw_{<3,0>} \parallel (Up!1;C)_{<2,1>}$
n_3
$Sw'_{<3,2>} \parallel C_{<2,2>}$
$(\sigma_{B,1}, \sigma_{B_1})$
$\mathbf{rcfg}(\mathbf{Up},\mathbf{ft})$
n_5
$Sw'_{<4,2>} \parallel C_{<2,2>}$
n_6
$Sw_{<3,0>} \parallel (Up!1;C)_{<2,1>}$

Figure 7: Symbolic execution of the SDN in (2); excerpt

$n_0 \to n_1 \to n_2 \to n_4$: instead of immediately installing the "drop everything" policy in SW', the network first forwards a regular packet from port 1 to port 2. The race is detected by the incomparable clocks $\langle 3,0\rangle$ and $\langle 2,1\rangle$ in n_4. Furthermore, the sequence associated packets $\sigma_{B,1}$ and $\sigma_{R,1}$ can be seen as the (minimal) explanation of the race. A similar reasoning holds for the race in $n_0 \to n_1 \to n_2 \to n_6$. Case (ii) corresponds to the symbolic execution $n_0 \to n_1 \to n_2 \to n_3$. Note that all vector clocks can be compared along this execution, so no data race is identified.

5 Tracer

Tracer [1] is the tool developed in this work. It exploits the symbolic semantics of DyNetKAT as described in Section 4, and computes minimal sets of packets that enable races between the control and data planes of an inputted SDN encoded in DyNetKAT. In this section we provide the algorithm behind Tracer, instructions on how to install and run the tool, and one example of using Tracer.

We define a race detection function $rd(SDN,k)$ in (5) that identifies minimal symbolic executions of SDN witnessing races up to a given depth k in the execution tree of SDN (as illustrated in Section 4). Furthermore, the function returns the network packets enabling these races, as explanations. The traced packets are encoded as α^i in (5).(d). Recall that every complete test $\alpha_i = (f_1 = v_1)\cdot\ldots\cdot(f_m = v_m)$ entails a unique packet $\{f_1 = v_1,\ldots,f_m = v_m\}$. The Tracer Algorithm 1 implements the $rd(-)$ function based on an interplay behind Python and Maude. Invoking Maude is for deriving DyNetKAT policies in head normal forms according to the DyNetKAT complete axiomatization in [5]. These head normal forms (denoted by $hnf(d_i)$ and $hnf(d_j)$ in (5).(d) and in (5).(e)) are further exploited for identifying packet forwarding steps within the analysed SDN ($\alpha^i \cdot \pi_{\alpha^i}$ in (5).(d)), or communication steps between the data and control planes ($\mathbf{rcfg}(\mathbf{X},\mathbf{p})$ based on $X_{\gamma^i}!p_{\gamma^i}$ and $X_{\gamma^j}?p_{\gamma^j}$ in (5).(e)). Observe that vector clocks are updated in accordance with the clocks of the symbolic rules: (5).(d) complies to ($\mathbf{Symb}_{\checkmark}$) and (5).(e) complies to ($\mathbf{Symb}_{||}$), respectively. The function $\textit{not-race}(\vec{C}_1,\ldots,\vec{C}_n)$ returns **true** whenever any two vector clocks $\vec{C}_i$ and $\vec{C}_j$ are incomparable, and **false** otherwise.

$$\begin{array}{ll}
(a) & rd(\Pi_{1\leq i\leq n} d_{i_{\vec{C}_i}}, 0) \triangleq \emptyset \\
(b) & rd(\Pi_{1\leq i\leq n} d_{i_{\vec{C}_i}}, k+1) \triangleq \\
(c) & \quad \textit{if not-race}(\vec{C}_1,\ldots,\vec{C}_n) \textit{ then return} \\
(d) & \qquad \bigcup\limits_{\substack{1\leq i\leq n \\ (\alpha^i\in A^i)\wedge(\alpha^i\cdot\pi_{\alpha^i};d_{\alpha^i}\in hnf(d_i))}} \alpha^i :: rd(d_{\alpha^i_{\vec{C}_i[i]++}} \,||\, \Pi_{\substack{1\leq j\leq n \\ j\neq i}} d_{j_{\vec{C}_j}}, k) \\
(e) & \qquad \bigcup\limits_{\substack{1\leq i\neq j\leq n \\ (\gamma^i\in\Gamma^i)\wedge(X_{\gamma^i}!p_{\gamma^i};d_{\gamma^i}\in hnf(d_i)) \\ (\gamma^j\in\Gamma^j)\wedge(X_{\gamma^j}?p_{\gamma^j};d_{\gamma^j}\in hnf(d_j)) \\ (X_{\gamma^i}==X_{\gamma^j}==X)\wedge(p_{\gamma^i}==p_{\gamma^j}==p)}} \mathbf{rcfg}(\mathbf{X},\mathbf{p}) :: rd(d_{\gamma^i\ \vec{C}_i[i]++} \,||\, d_{\gamma^j\ max(\vec{C}_i[i]++,\vec{C}_j)[j]++} \,||\, \Pi_{\substack{1\leq j\leq n \\ j\neq i}} d_{j_{\vec{C}_j}}, k) \\
(f) & \textit{else return } \downarrow
\end{array} \tag{5}$$

We use $\downarrow$ in (5).(f) as a marker symbol indicating that $rd(-)$ identified a race witnessing trace. Every trace ending with $\downarrow$ returned by $rd(SDN,k)$ encodes a set of packets witnessing concurrent behaviour within the SDN. We use :: in (5) as a constructor (concatenation) for such witnesses.

From an algorithmic perspective: (5).(d) is handled in lines $13-21$ of Algorithm 1, whereas (5).(e) is handled in lines $22-31$. Note that the aforementioned head normal forms in (5) are computed using the DyNetKAT axiomatization implemented in Maude [5]: lines $8, 15, 24$ and 25 in Algorithm 1. Line 8 invokes the application of a Maude-defined "projection" operator `pi{m}` that unfolds the given expression N_i up to depth `m`. Checking for deadlock in line 12 of Algorithm 1 is a stopping condition based on whether all parallel components in `curr` are either $\perp$ or start with communication actions that cannot be matched by any other component. Lines $33-34$ extract the race witnessing packets, in a post-processing step.

Tracer at Work. Tracer is publicly available at [1]. A complete installation guide can be found in README.md. Requirements for running Tracer include a Linux operating system, specifically Ubuntu 20.04[1] with Python (version $> 3.10.12$). The tool also uses Maude 3.1, that is included in the installation of Tracer. To use it, run the command in the following form:

```
> python tracer\_runner.py <path_to_maude> <path_to_model_in_maude>
```

The command has several optional parameters as given in Table 1. Note that the parameters with values should be inputted without the space between the parameter and the value. For example, write `-grace` to produce only graphs and traces witnessing data races in the provided model.

Table 1: Tracer command line

Parameter	Value	Explanation
-c	-	output text with color
-t	-	show tracing steps
-u	int	unfold depth
-g	'race' or 'full'	types of trees and traces to generate (race witnesses only, or full trees/traces)
-f	string	set a name for text output file (copy of console output)

SDN Encoding in Tracer: Example. The DyNetKAT encoding in (2) is provided as input for Tracer in a Maude-compatible format as shown in Listing 1. The DyNetKAT recursive variables SW, SW' and

[1]Other Linux distributions might work, however, the development and testing were done on the specified version of Ubuntu.

```
1: **Input:** SDN with $k$ components as a DynetKAT model $(N_1 \parallel N_2 \parallel \ldots \parallel N_k)$
2: **Input:** Depth $m$ of the search
3: **Output:** The smallest sets of network packets enabling races in the SDN
4: for $i \in \{1,\ldots,k\}$ do
5:    Initialize vector clocks $C_i = \langle 0,\ldots,0\rangle$ of size $k$
6: end for
7: for $i \in \{1,\ldots,k\}$ do
8:    Let $pmN_i$ be the result of invoking Maude > reduce $pi\{m\}(N_i)$
9: end for
10: Initialize curr with $(pmN_1,C_1) \parallel (pmN_2,C_2) \parallel \ldots \parallel (pmN_k,C_k)$
11: Initialize symb-traces with $\emptyset$
12: if not-deadlock(curr) then
13:    for each element $(N_i,C_i)$ at position $i$ in curr do
14:       if (complete-test-assignment;$d_i$) is a summand of $N_i$ then
15:          Let $rd_i$ be the result of invoking Maude > reduce $d_i$
16:          Set curr to $(N_1,C_1) \parallel \ldots \parallel (rd_i,C_i') \parallel \ldots \parallel (N_k,C_k)$
17:          where $C_i'$ is $C_i$ incremented at position $i$
18:          Append (complete-test, $(C_1,\ldots,C_i',\ldots,C_k)$) to symb-traces
19:          go to step 12
20:       end if
21:    end for
22:    for all pairs of elements $(N_i,C_i)$ and $(N_j,C_j)$ at positions $i$ and $j$ in curr do
23:       if $(X!p;d_i)$ is a summand of $C_i$ and $(X?p;d_j)$ is a summand of $C_j$ then
24:          Let $rd_i$ be the result of invoking Maude > reduce $d_i$
25:          Let $rd_j$ be the result of invoking Maude > reduce $d_j$
26:          Set curr to $(N_1,C_1) \parallel \ldots \parallel (rd_i,C_i') \parallel \ldots \parallel (rd_j,C_j') \parallel \ldots \parallel (N_k,C_k)$
27:          where $C_i'$ is $C_i$ incremented at position $i$, and $C_j'$ is max$(C_i',C_j)$ incremented at position $j$
28:          Append (rcfg$(X,p)$, $(C_1,\ldots,C_i',\ldots,C_j',\ldots,C_k)$) to symb-traces
29:          go to step 12
30:       end if
31:    end for
32: end if
33: Let races be a set of sets of packets, initialized with $\emptyset$
34: for all symb-trace in symb-traces do
35:    if a prefix s-tr' of symb-trace ends with incomparable vector clocks $(C_1,\ldots,C_k)$ then
36:       Extract all packets pkt based on every max-test in s-tr'
37:       Add the set of packets pkt to races
38:    end if
39: end for
40: return races
```

Algorithm 1: Detecting races in SDN using DyNetKAT models

C in (2) are declared as the constants `SW`, `SWP` and `C` of `Recursive` type in Listing 1. The operator `getRecPol(...)` is a syntactic wrapper around these recursive operators. The actual definitions of the switch and controller follow closely the syntax in (2). The communication channels $Help$ and Up translate to the constants `Help` and `Up` of type `Channel` in Maude. The DyNetKAT non-deterministic choice $\oplus$ translates to `o+` in Maude. The DyNetKAT constants $\mathbf{0}$ and $\bot$ are mapped to `zero` and `bot`. The entire SDN consisting of the switch SW and controller C as in Figure 3 is defined by the constant `Init` of type `DNA` in Maude. Note that the NetKAT expressions encoding the forwarding policies are provided as strings in Maude; e.g., `"(flag = regular).(pt = 1).(pt <- 2)"`. The model in Listing 1 along with the depth k of the analysis are provided as input to Tracer.

```
fmod MODEL is
    [...]

    ops Init : -> DNA .
    ops SW, SWP, C : -> Recursive .
    ops Help, Up : -> Channel .

    eq getRecPol(SW) =
         "(flag = regular) . (pt = 1) . (pt <- 2)" ; SW o+
         "(flag = blocking) . (pt = 1) . 1" ;( (Help ! "one") ; SW ) o+
         (Up ? "one") ; SWP .
    eq SWP = zero ; bot .
    eq getRecPol(C) = (Help ? "one") ; ( (Up ! "one") ; C ) .

    eq Init = C || SW .
endfm
```

Listing 1: Maude encoding of the SDN in (2)

Tracer Output: Example. Figure 8 showcases the output races as identified by Tracer, in a graphical format. (We use fl, B and R as shorthand for $flag, blocking$ and $regular$, respectively.) The sequence of nodes $0 \to 1 \to 3 \to 5$ corresponds to the symbolic execution $n_0 \to n_1 \to n_2 \to n_4$ in Figure 7, encoding a race. The sequence of nodes $0 \to 1 \to 3 \to 6$ in Figure 8 corresponds to the symbolic execution $n_0 \to n_1 \to n_2 \to n_6$ Figure 7, encoding a race as well. Furthermore, the labels along these executions are minimal explanations of how the races can be enabled. Note how the corresponding clocks in Figure 8 match their counterparts in Figure 7.

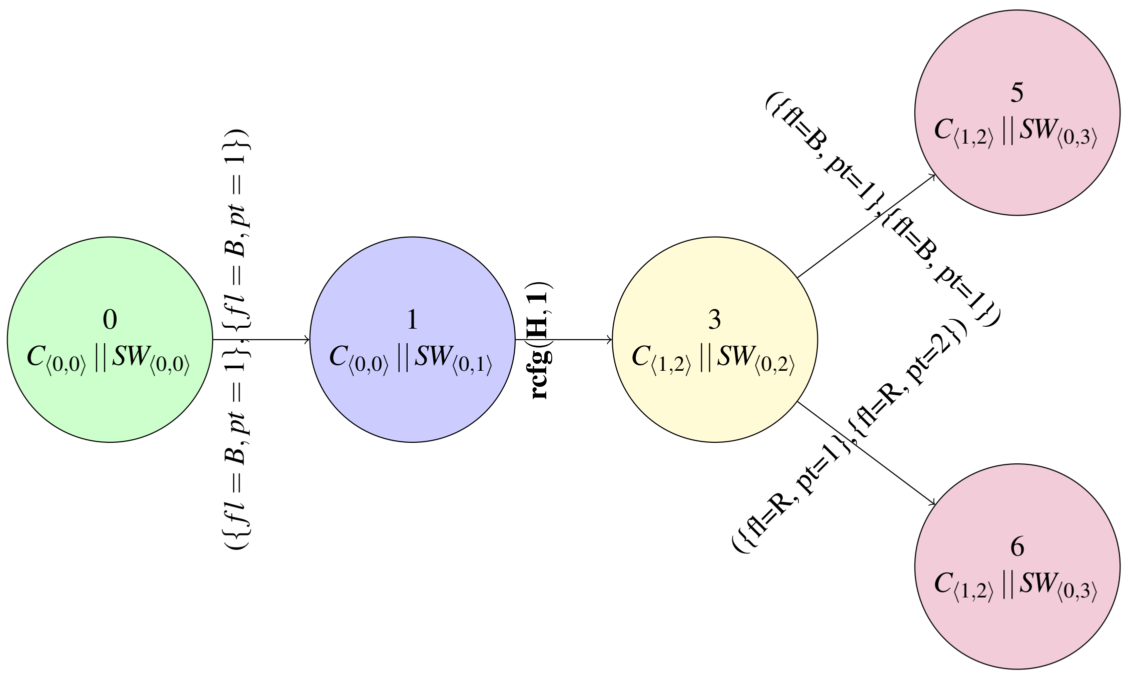

0
$C_{\langle 0,0\rangle} \,||\, SW_{\langle 0,0\rangle}$
1
$C_{\langle 0,0\rangle} \,||\, SW_{\langle 0,1\rangle}$
3
$C_{\langle 1,2\rangle} \,||\, SW_{\langle 0,2\rangle}$
5
$C_{\langle 1,2\rangle} \,||\, SW_{\langle 0,3\rangle}$
6
$C_{\langle 1,2\rangle} \,||\, SW_{\langle 0,3\rangle}$
$(\{fl = B, pt = 1\}, \{fl = B, pt = 1\})$
rcfg(H,1)
({fl=B, pt=1},{fl=B, pt=1})
({fl=R, pt=1},{fl=R, pt=2})

Figure 8: Races in $SW \,||\, C$ up to depth 3

Tracer outputs the witnesses of data races in a textual format as well, as illustrated in Figure 9. These traces can be in short form, encoding the input packets and/or reconfiguration steps within symbolic executions without vector clocks. Traces in long form show the action performer (switch `SW`, controller `C` or a handshake between the switch and the controller `SW -> C`), the vector clocks (`[0, 0]`, `...`), and the corresponding node ID in the graph as well.

6 Conclusions

In this paper, we introduced Tracer [1], a tool for detecting and explaining data races in SDNs as defined in [4]. These systems exhibit concurrent behavior due to the interaction between data plane processing, and dynamic reconfigurations between the data and control planes. Tracer focuses on pin-pointing data races in SDN models encoded within the DyNetKAT [5] framework. In addition, Tracer provides explanations of how these data races can be enabled by identifying sequences of packets which, whenever fed to the SDN under analysis, lead to concurrency between the data and control planes. The tool is built on top of the DyNetKAT axiomatisation implemented in Maude. In the future, we plan to analyze Tracer's performance on benchmarks with larger SDN models. Additionally, we aim to implement a parallelized version of Tracer to improve its efficiency.

Acknowledgements. This work was supported by the project ZORRO, no. KICH1.ST02.21.003 of the research programme Key Enabling Technologies (KIC) which is (partly) financed by the Dutch Research Council (NWO).

```
RACE SHORT TRACES
Trace 0:
"(flag = blocking) . (pt = 1) "; rcfg('Help', '"one"');
"(flag = blocking) . (pt = 1)"

Trace 1:
"(flag = blocking) . (pt = 1) . 1"; rcfg('Help', '"one"');
"(flag = regular) . (pt = 1)"



RACE LONG TRACES
Trace 0:
{C[0, 0] || SW[0, 0]} nid:0;
[SW] "(flag = blocking) . (pt = 1) " {C[0, 0] || SW[0, 1]} nid:1;
[SW -> C] rcfg('Help', '"one"') {C[1, 2] || SW[0, 2]} nid:3;
[SW] "(flag = blocking) . (pt = 1)" {C[1, 2] || SW[0, 3]} nid:5;


Trace 1:
{C[0, 0] || SW[0, 0]} nid:0;
[SW] "(flag = blocking) . (pt = 1) " {C[0, 0] || SW[0, 1]} nid:1;
[SW -> C] rcfg('Help', '"one"') {C[1, 2] || SW[0, 2]} nid:3;
[SW] "(flag = regular) . (pt = 1) " {C[1, 2] || SW[0, 3]} nid:6;
```

Figure 9: Race traces of *SW* || *C* with unfold 3

References

[1] (2024): *Tracer*. https://github.com/EZUTwente/DyNetiKAT_with_race_tracing/blob/master/README.md.

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger & David Walker (2014): *NetKAT: semantic foundations for networks*. In Suresh Jagannathan & Peter Sewell, editors: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, ACM, pp. 113–126, doi:10.1145/2535838.2535862. Available at `https://doi.org/10.1145/2535838.2535862`.

[3] Jos C. M. Baeten & W. P. Weijland (1990): *Process algebra*. *Cambridge tracts in theoretical computer science* 18, Cambridge University Press.

[4] Georgiana Caltais & Hossein Hojjat (2024): *Symbolic Race Identification in DyNetKAT*. `https://drive.google.com/file/d/1G46mSe7-Xr-b6Aq85wW-7FgKPXr0Ecn5/view`.

[5] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi & Hünkar Can Tunç (2022): *DyNetKAT: An Algebra of Dynamic Networks*. In Patricia Bouyer & Lutz Schröder, editors: *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, *Lecture Notes in Computer Science* 13242, Springer, pp. 184–204, doi:10.1007/978-3-030-99253-8_10. Available at `https://doi.org/10.1007/978-3-030-99253-8_10`.

[6] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever & Martin T. Vechev (2016): *SDNRacer: concurrency analysis for software-defined networks*. In Chandra Krintz & Emery D. Berger, editors: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, ACM, pp. 402–415, doi:10.1145/2908080.2908124. Available at `https://doi.org/10.1145/2908080.2908124`.

[7] James Hamilton (2009): *Networking: The last bastion of mainframe computing*. Available at `https://perspectives.mvdirona.com/2009/12/networking-the-last-bastion-of-mainframe-computing/`.

[8] Dexter Kozen (1997): *Kleene Algebra with Tests*. *ACM Trans. Program. Lang. Syst.* 19(3), pp. 427–443, doi:10.1145/256167.256195. Available at `https://doi.org/10.1145/256167.256195`.

[9] Leslie Lamport (2019): *Time, clocks, and the ordering of events in a distributed system*. In Dahlia Malkhi, editor: *Concurrency: the Works of Leslie Lamport*, ACM, pp. 179–196, doi:10.1145/3335772.3335934. Available at `https://doi.org/10.1145/3335772.3335934`.

[10] Xiaoye Steven Sun, Apoorv Agarwal & T. S. Eugene Ng (2015): *Controlling Race Conditions in OpenFlow to Accelerate Application Verification and Packet Forwarding*. *IEEE Transactions on Network and Service Management* 12(2), pp. 263–277, doi:10.1109/TNSM.2015.2419975.

[11] Andrew S. Tanenbaum & Maarten Van Steen (2007): *Distributed Systems: Principles and Paradigms*, 2nd edition. Pearson Prentice Hall, Upper Saddle River, NJ. Available at `https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28GÃűschka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf`.

[12] Evgenii M. Vinarskii, Jorge López, Natalia Kushik, Nina Yevtushenko & Djamal Zeghlache (2019): *A Model Checking Based Approach for Detecting SDN Races*. In Christophe Gaston, Nikolai Kosmatov & Pascale Le Gall, editors: *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings*, *Lecture Notes in Computer Science* 11812, Springer, pp. 194–211, doi:10.1007/978-3-030-31280-0_12. Available at `https://doi.org/10.1007/978-3-030-31280-0_12`.

[13] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang & Guofei Gu (2017): *Attacking the Brain: Races in the SDN Control Plane*. In Engin Kirda & Thomas Ristenpart, editors: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, USENIX Association, pp. 451–468. Available at `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-lei`.

Static Analysis Framework for Detecting Use-After-Free Bugs in C++

Vlad-Alexandru Teodorescu
Faculty of Computer Science
Alexandru Ioan Cuza University of Iasi, Romania
teodorescu.vlad@yahoo.com

Dorel Lucanu
Faculty of Computer Science
Alexandru Ioan Cuza University of Iasi, Romania
dorel.lucanu@gmail.com

Pointers are a powerful, but dangerous feature provided by the C and C++ programming languages, and incorrect use of pointers is a common source of bugs and security vulnerabilities. Making secure software is crucial, as vulnerabilities exploited by malicious actors not only lead to monetary losses, but possibly loss of human lives. Fixing these vulnerabilities is costly if they are found at the end of development, and the cost will be even higher if found after deployment. That is why it is desirable to find the bugs as early in the development process as possible. We propose a framework that can statically find use-after-free bugs at compile-time and report the errors to the users. It works by tracking the lifetime of objects and memory locations pointers might point to and, using this information, a possibly invalid dereferencing of a pointer can be detected. The framework was tested on over 100 handwritten small tests, as well as 5 real-world projects, and has shown good results detecting errors, while at the same time highlighting some scenarios where false positive reports may occur. Based on the results, it was concluded that our framework achieved its goals, as it is able to detect multiple patterns of use-after-free bugs, and correctly report the errors to the programmer.

1 Introduction

Pointers are a powerful, but dangerous feature provided by the C and C++ programming languages, and incorrect use of pointers is a common source of bugs and security vulnerabilities. Most new languages lack pointers or severely restrict their capabilities, thus eliminating these problems and providing memory safety. Nonetheless, many C & C++ programs work with pointers safely and they are still considered a very useful feature of the language. Programmers who safely work with pointers maintain an internal model of when memory accessed through those pointers should be allocated and subsequently freed. Commonly applied models include garbage collection, Resource Acquisition Is Initialization (RAII), and smart pointers.

However, because the chosen model is frequently not documented in the program, it might not be well understood when modifying the source code, and errors can appear. These can lead to various problems such as program crashes, unpredictable behavior, or security vulnerabilities. Some infamous examples of critical vulnerabilities caused by memory bugs are:

- Heartbleed (CVE-2014-0160) [7] - exposed secrets in the popular OpenSSL library. It affected over 66% of the active sites on the Internet.
- BlueKeep (CVE-2019-0708) [5] - Remote Code Execution in Microsoft's RDP. It affected all unpatched Windows versions.
- EternalBlue (CVE-2017-0144) [6] - Remove Code Execution in Windows Microsoft's SMB. It affected all unpatched Windows versions. It was used to carry out cyber attacks which caused major damage, like WannaCry [14] or NotPetya [13]

Making secure software is crucial, as vulnerabilities exploited by malicious actors not only lead to monetary losses, but possibly loss of human lives [20]. Fixing these vulnerabilities is costly if they are found at the end of development, and the cost will be even higher if found after deployment. That is why it is desirable to find the bugs as early in the development process as possible. One class of tools that can help programmers identify software defects early is static analysis tools. These tools analyze source code without executing it, and can even point out bugs while the programmer is still writing the code.

One common bug in C++ that can cause vulnerabilities is the use-after-free pattern. They occur when a program continues to use a pointer after the memory it points to has been freed. This usually causes program crashes, but in the worst cases, it may lead to critical vulnerabilities such as those mentioned earlier.

Identifying this type of bug is not trivial, as C++ is a complex language that gives the programmer a lot of control over memory, without many constraints. Access to on-demand allocation and deallocation of memory, combined with pointer arithmetic and the existence of functions that can affect memory outside of their scope make tracking pointer operations at compile time difficult, and sometimes even impossible. In addition to the difficulty of identification by static analyzers, this type of bugs are also very hard to identify during code reviews, even by experienced programmers.

Contribution This paper proposes a framework intended to provide sound static analysis by using semantic information that can be inferred from the source code and some information explicitly provided by the programmer. In particular, the framework aims to track the lifetime of all memory regions referenced by pointers during the program's execution and ensure that dereferenced pointers' pointed-to memory is valid.

Firstly, a simplified model is extracted from the target program to facilitate the analysis. Types and instructions are separated into multiple categories depending on their properties and possible effects on memory. Then, the identification of errors is done through dataflow analysis on the control flow graph of the simplified program. The use of tools from the Clang ecosystem facilitates all these steps.

Paper Structure This paper is split into 4 sections, each describing theoretical aspects or implementation details of the framework. Section 2 describes other approaches for detecting memory bugs. Section 3 presents all the steps of the analysis process used for detecting and reporting the errors. Section 4 describes some of the implementation details of the steps presented in the previous section and the results of several experiments conducted to evaluate the framework's performance.

2 Related Work

Since software security is crucial, there have been many projects that try to detect and prevent vulnerabilities before they are available to the public and can be exploited. The classes of bugs detected and the concepts used to detect them differ from project to project, but most of them fall under 3 main categories: static analyzers, dynamic analyzers, and hybrid approaches

2.1 Static Analyzers

Static analyzers are tools that check different properties of programs by only looking at their source code, without the need to execute them.

Herb Sutter describes in [11] an approach to identify use-after-free bugs in C++ through static analysis and is the main inspiration for this framework and paper.

Multiple other tools aim to find different types of memory bugs through static analysis. The Clang static analyzer supplements the built-in warnings of the compiler with the same name with over 100 powerful checkers [3] that try to detect both general C++ bugs, as well as application-specific ones. The other 2 main C++ compilers, GNU GCC and Microsoft Visual C++ have similar static analysis tools.

In addition to the warnings produced by the compilers, IDEs have their own tools looking for errors. Jetbrains CLion provides tens of different warnings [4] and also integrates with other static analysis tools in order to have as much coverage as possible when looking for bugs. Other popular IDEs such as Microsoft Visual Studio or XCode have similar approaches.

2.2 Dynamic Analyzers

Dynamic analysis is the process of evaluating a program by executing it and observing its behavior. In the case of C++ this usually involves tracking memory management and access, as well as concurrency, to ensure no errors are present.

Valgrind [16] is a powerful tool suite that is primarily used for detecting memory management and threading bugs in C++ applications. It includes various tools that can identify memory leaks, invalid memory access or mismanagement of memory. Valgrind emulates the execution of programs, so it has complete control over the low-level operations. Because of this, it is able to provide detailed information about potential errors in the program, as well as their causes.

Address Sanitizer [18] is another popular dynamic analyzer that can detect memory errors. It employs a specialized memory allocator and code instrumentation to accurately detect bugs at their point of occurrence. It is currently the most widely used tool due to it having the smallest performance impact, while still being accurate. It is also integrated in all major C++ compilers.

2.3 Hybrid Approaches

There are a few tools that take a hybrid approach by combining both static and dynamic analysis, and one of the most popular and efficient uses of this is fuzz testing [19]. A fuzzer uses both static analysis to look at the source code, and dynamic analysis to observe the flow of execution in order to generate sets of inputs that cover as many branches as possible. One of the more popular fuzzers is AFL++ [9]. Then, these inputs are fed to the program, and dynamic analyzers are used to find possible errors during the execution. The wide coverage of inputs generated by the fuzzer helps find errors that happen very rarely, which may not be caught when using a hand-written set of tests.

3 The Proposed Framework

The analysis process consists of several steps that will be described in this section. Firstly, the language used to describe the model extracted from the input C++ program is defined. Next, the operations that transform a C++ program into the form used by the framework are presented. Finally, we describe the analysis process, what information it uses, and how it is obtained.

3.1 The Language

The first step to facilitate the static analysis is to extract from a C++ program a model expressing how the program interacts with memory. The language we will use is based on C++ [12], with some modifications to the operations that manipulate memory. The additional syntax needed to manipulate the object's lifetime is described in Figure 1.

$$
\begin{array}{rcll}
\langle\text{instructions}\rangle & ::= & \ldots & \\
& | & \langle\text{var}\rangle = \texttt{allocate}(\langle\text{exp}\rangle); & \text{allocation} \\
& | & \langle\text{var}\rangle = [\langle\text{exp}\rangle]; & \text{lookup} \\
& | & [\langle\text{exp}\rangle] = \langle\text{exp}\rangle; & \text{mutation} \\
& | & \texttt{deallocate}(\langle\text{exp}\rangle); & \text{deallocation} \\
& | & \texttt{create}(\langle\text{var}\rangle, \langle\text{type}\rangle); & \text{variable creation} \\
& | & \texttt{destroy}(\langle\text{var}\rangle); & \text{variable destruction} \\
\langle\text{type}\rangle & ::= & \texttt{Owner} & \\
& | & \texttt{Pointer} & \\
& | & \texttt{Value} &
\end{array}
$$

Figure 1: Additional syntax for handling the lifetime of objects

The computational state contains two components: a store, which maps variables into addresses, and the memory, which maps addresses into values.

$$
\begin{aligned}
\text{Values} &= \text{Integers} \cup \text{Atoms} \cup \text{Addresses} \\
\text{Memory} &= \bigcup_{\text{A} \subseteq \text{Addresses}} (\text{A} \to \text{Values}) \\
\texttt{null} &\in \text{Atoms} \\
\text{Stores}_V &= \text{V} \to \text{Addresses} \\
\text{States}_V &= \text{Stores}_V \times \text{Memory}
\end{aligned}
$$

where V is a finite set of variables.

The behavior of the new instructions is defined by a transition relation $\leadsto$ between configurations, which can be:

- **nonterminal:** an instruction-state pair $\langle i, (s,m)\rangle$, where (s,m) is a State, $FV(i) \subseteq dom(s)$ ($FV(i)$ is the set of free variables in instruction i);
- **terminal:** a state (s,m) or **error**.

The semantics of the new instructions is defined below. Here $[f|x:y]$ denotes the function that maps x into y and all other $x' \in dom(f)$ into $f(x')$. The notation $f|_S$ means the restriction of the function f to the domain S. $dom(f)$ is the domain of the function f and $val(e)$ is the value of expression e.

- Allocation:
  $\langle v = \texttt{allocate}(e), (s,m)\rangle \leadsto (s, [m|s(v) : a|a : null|a+1 : null|\ldots|a+val(e)-1 : null])$
  where $a, a+1, \ldots, a+val(e)-1 \in \text{Addresses} - dom(m)$
- Lookup:
  $\langle v = [e], (s,m)\rangle \leadsto (s, [m|s(v) : m(val(e))])$ if $val(e) \in dom(m)$
  $\langle v = [e], (s,m)\rangle \leadsto \textbf{error}$ if $val(e) \notin dom(m)$

- Mutation:
  $\langle [e] = e', (s,m)\rangle \rightsquigarrow (s, [m|val(e) : val(e')])$ if $val(e) \in dom(m)$
  $\langle [e] = e', (s,m)\rangle \rightsquigarrow$ **error** if $val(e) \notin dom(m)$
- Deallocation:
  $\langle \texttt{deallocate}(e), (s,m)\rangle \rightsquigarrow (s, m|_{dom(m)-A})$
  if $val(e)$ is the first address in a set of addresses returned by an `allocate` instruction before the call to `deallocate` and $val(e) \in dom(m)$
  $\langle \texttt{deallocate}(e), (s,m)\rangle \rightsquigarrow$ **error** otherwise

  where A is the set of Addresses returned by the `allocate` instruction.
- Variable Creation:
  $\langle \texttt{create}(v,t), (s,m)\rangle \rightsquigarrow ([s|v : a], [m|a : null])$
  where $a \in$ Addresses $- dom(m)$;
- Variable Destruction:
  $\langle \texttt{destroy}(v), (s,m)\rangle \rightsquigarrow (s|_{dom(s)-v}, m|_{dom(m)-s(v)}$

The allocation instruction activates and initializes the required cells in the heap. The only requirement for these cells is that they were previously inactive and are consecutive. The starting address is unspecified.

The Lookup, Mutation, and Deallocation operations cause memory errors (indicated by the terminal configuration **error**) whenever an invalid address is dereferenced or deallocated. This would correspond to a crash when running a C++ program.

The `create` and `destroy` instructions are used to emulate the behavior of stack variable allocation and deallocation when entering or exiting a scope in C++.

Types There are 3 classes of types based on their properties: `Owner`, `Pointer`, `Value`.

An `Owner` is a variable that owns a zone of memory. This means it can use all four memory management functions from our language on the memory it owns. During the variable's creation, it allocates some memory and during its destruction it deallocates it. Some functions may alter the `Owner`, thus changing the memory zone managed by the variable to another one. Dereferencing an `Owner` is always valid. (for example, a type classified as Owner is `std::vector`)

A `Pointer` is a variable that points to some memory it does not own. It does not allocate or deallocate any memory. This means it can only use the Lookup and Mutate instructions on the memory it points to. All use-after-free errors happen when dereferencing this class of variables, as they can still point to a memory zone that has already been deallocated by an `Owner`. A `Pointer` that produces an error when dereferenced is called invalid.

A variable is classified as a `Value` if it is neither an `Owner`, nor a `Pointer`. All variables that do not interact with heap memory fall into this category.

Annotations In addition to the modifications to instructions, we also extend the function definition syntax to be able to make annotations that represent preconditions about the lifetime of the input or output parameters of the function and postconditions about the lifetime of its outputs.

We define the lifetime of a memory zone as the sequence of instructions in the program during which it is valid to dereference it. Formally, the address a is alive if none of its lookup, mutation, or deallocation transition to the **error** configuration.

The lifetime of a memory zone begins when the `allocate` instruction activates the corresponding memory addresses and ends when the `deallocate` instruction deactivates them.

The syntax of the annotations is:

$$
\begin{array}{rcll}
\langle\text{annotated\_func}\rangle & ::= & \langle\text{annotation}\rangle\langle\text{func\_definition}\rangle & \\
\langle\text{annotation}\rangle & ::= & \texttt{pre}((\langle\text{var}\rangle,\{\langle\text{lifetime}\rangle*\})+) & \text{precondition} \\
 & | & \texttt{post}((\langle\text{var}\rangle,\{\langle\text{lifetime}\rangle*\})+) & \text{postcondition} \\
\langle\text{lifetime}\rangle & ::= & \langle\text{var}\rangle & \\
 & | & \texttt{null} & \\
 & | & \texttt{global} & \\
 & | & \texttt{invalid} &
\end{array}
$$

Let p be a function parameter with the precondition $\texttt{pre}(p,S)$ attached. This means that p must have the same lifetime as at least one element in S when the function is called. For example, when calling the function defined in listing 1, the lifetime of the parameter z has to be equal to either that of x or y, otherwise, an error is raised.

To attach a postcondition to the return value of a function, we will use the function's name. Let f be a function with postcondition $\texttt{post}(f,S)$. This means that the return value of f must have the same lifetime as at least one element in S. For example, the value returned by the function defined in listing 2 has to have a lifetime equal to that of x, or be null.

Listing 1: A function with a precondition

```
pre(z,{x,y})
Pointer f(Pointer x,Pointer y,
Pointer z)
{
...
}
```

Listing 2: A function with a postcondition

```
post(g, {x,null})
Pointer g(Pointer x)
{
...
}
```

3.2 Transforming a C++ Program

We define a set of rules that will help the framework transform the input C++ program into our proposed simplified language.

3.2.1 Variable Types

A variable is classified as an `Owner` if its type satisfies any of the following conditions:

- it satisfies the standard Container requirements and has a user-provided destructor - for example, `std::vector`;
- it provides the dereference operator and has a user-provided destructor - for example, `std::unique_ptr`;
- it has a data member or base class of type `Owner`.

A variable is classified as a `Pointer` if its type is not an `Owner` and satisfies any of the following conditions:

- it satisfies the standard Iterator requirements;
- Is trivially copyable, is copy constructible and assignable, and provides the dereference operator.

- it has a data member or base class of type `Pointer`;
- it is a capture by reference inside a lambda function.

A variable is classified as a `Value` if it is neither an `Owner` nor a `Pointer`. All variables that do not interact with heap memory fall into this category.

3.2.2 Instructions

Control-flow instructions, assignments, expressions, and most other instructions remain unchanged. The only cases where explicit transformations are needed are memory-related instructions.

Raw pointer arithmetic makes static tracking of memory zones variables are pointing to very difficult, sometimes even impossible. That is why our framework forbids it, as the analysis may give imprecise, or even wrong results because of these instructions.

Manual memory allocation through `new` is considered a special case of creating a global `Owner` variable. Manual memory deallocation through `delete` corresponds to the destruction of this variable.

```
                    create (global_owner, Owner);
p = new int;  ⟶    global_owner = allocate (4);
                    p = global_owner;
```

```
                destroy (global_owner);
delete p;  ⟶   deallocate (global_owner);
```

In C++, when a scope begins, all local variables are implicitly allocated on the stack. In a similar manner, when a scope ends, all local variables are implicitly deallocated. In our programming language, this is done explicitly.

```
               {
{                create (x, Value);
  int x;   ⟶    ...
  ...            destroy (x);
}              }
```

3.2.3 Functions

Functions can be explicitly annotated by the programmer with preconditions and postconditions about the lifetime of their `Pointer` parameters. If these are not explicitly mentioned, then the following default conditions will be enforced:

Precondition No `Pointer` arguments can point to a non-const global `Owner` or a local `Owner` being passed by a non-const reference to the same function call. Also, no two `Pointer` arguments should point to the same non-const `Owner`.

Postcondition If the function returns a `Pointer` variable, its lifetime should be at least as long as one of the arguments.

Example program

Following the rules above, an example transformation of a section of a program is presented in listings 3 and 4.

Listing 3: Example C++ program

```
int x;
int* p;
...
if (x == 2) p = &x;
else {
    int y;
    p = &y;
}
*p = 3;
```

Listing 4: Transformed C++ program

```
create(x, Value);
create(p, Pointer);
...
if (x == 2) p = &x;
else {
    create(y, Value);
    p = &y;
    destroy(y);
}
*p = 3;
```

3.3 Analysis

After the steps above, the framework will perform static intraprocedural analysis, on each function definition, to enforce the following rules:

1. It is an error to dereference an invalid `Pointer`.
2. It is an error to dereference an `Owner` that was moved from (transferred ownership to another variable).
3. It is an error to use raw pointer arithmetic.
4. It is an error to pass a `Pointer` as a function argument if it is invalid, or violates the preconditions of the function.
5. It is an error to return a `Pointer` from a function that is invalid or violates the postconditions of the function.

To enforce rule 3., we will always report a possible error when pointer arithmetic is present in the analyzed program. Let *p* be a pointer and *v* another variable. The following expressions will produce an error:

- p+v
- v+p
- p++
- ++p
- p-v
- v-p
- p- -
- - -p
- p+=v
- p-=v
- p[x]
- x[p]

While rule 3. can be easily enforced through the analysis of the AST of the program and the types of variables, the rest of the rules need more complex, path-sensitive analysis. They all result in the same behavior of the program: reaching the **error** configuration after a Lookup of Mutate instruction.

A way to enforce these rules is by maintaining all possible locations a pointer might point to, and whether they are valid or not. We will use the notions of *points-to-map* (pmap) and *points-to-set* (pset) to maintain this information.

The *points-to-set* of a variable *v* at instruction *i* in the program is the set of all possible memory zones it may point to at this moment during the execution of the program. This set can contain any of these elements:

var - this means that v currently points to the address of local variable;

global - v currently refers to a static variable or a memory owned by a const static `Owner`;

o' - v points to the address of an object owned by `Owner` o;

o" (etc.) - v refers to an object owned by an object owned by o. This is used in the case of hierarchies of `Owner` objects;

null - v currently is null;

invalid - v points to invalid memory.

The *points-to-map* at instruction i is a mapping between local variables that exist at the current moment in execution and their psets.

A variable v is considered invalid at instruction i if `invalid` $\in pmap_i(v)$. (see for example p on the last line in listings 3 and 4)

We have chosen to calculate $pmap_i$ for all instructions using Dataflow Analysis, as some form of path-sensitive analysis is required to correctly determine the points-to-sets.

3.4 Instantiation of the Dataflow Analysis Framework

Let $\mathscr{F} = (\mathscr{V}, \mathscr{E}, e)$ be the control flow graph of the program we want to analyze. To calculate all the points-to-maps at each node in the graph, we will solve the following dataflow system [17]:

- $L = \mathscr{V}$. The program labels are the labels of the nodes.
- $E = e$. We will start the analysis from the initial nodes of the CFG.
- $F = \mathscr{E}$. The flow of the analysis is determined by the edges in the graph.
- $D = \text{Variables} \rightharpoonup 2^{\text{PsetEntries}}$ is the analysis domain, where $\text{PsetEntries} = \text{Variables} \cup \text{Variables'} \cup \{null, invalid, global\}$, and $\text{Variables'} = \{v', v'', \ldots | v \in \text{Variables}\}$.
  The information we maintain is the points-to-map at each node.
- $\sqsubseteq$ is the partial order defined as follows: $f \sqsubseteq g$ iff $dom(f) \subseteq dom(g)$ and for all $v \in dom(f)$, $f(v) \subseteq g(v)$.
- $\perp = \emptyset$ is the smallest element, and the initial value associated with e is $\iota = \emptyset$.

The transfer function is the most important part of this system and affects the *pmap* in different ways depending on the instruction in the node.

φ_i is the transfer function corresponding to instruction i and takes as input d which is the union of all *pmaps* of predecessor nodes in the CFG and produces a new *pmap* after the effects of instruction i.

At different moments during the analysis, *pmap* entries will be invalidated. We define the invalidation of a set of variables S as

$$\text{invalid}_S(d) = [d | x : (d(x) - \{v, v', \ldots\}) \cup \{invalid\}]$$

where x such that $d(x) \cap \{v, v', \ldots\} \neq \emptyset$ for all $v \in S$.

Variable creation - when creating a variable, a new entry is added to the *pmap*. Let i be the instruction `create`(v,t). Then,

$$\varphi_i(d) = \begin{cases} [d|v : \{v'\}] & \text{if } t = \texttt{Owner} \\ [d|v : \{invalid\}] & \text{if } t = \texttt{Pointer} \\ d & \text{if } t = \texttt{Value} \end{cases}$$

Variable destruction - when destroying a variable, its entry is removed, and all other *pset*s that may refer to it are invalidated. Let i be the instruction `destroy`(v). Then,

$$\varphi_i(d) = \text{invalid}_v(d|_{dom(d)-\{v\}})$$

Mutation of owner - when a non-const use of a local `Owner` occurs, we will invalidate all pointers to it, making the conservative assumption that any use might cause reallocation. Let i be $f(o)$. Then,

$$\varphi_i(d) = \text{invalid}_o(d)$$

Copying - when a copy happens, all the entries in the destination's *pset* will be replaced by all the entries in the source's set. Let i be the instruction be $v = u$. Then,

$$\varphi_i(d) = [d|v : d(u)]$$

Moving - when a move happens, all the entries in the destination's *pset* will be replaced by all the entries in the source's set. In addition to this, the source's *pset* will be invalidated. Let i be the instruction be $v = std :: move(u)$. Then,

$$\varphi_i(d) = [d|v : d(u)|u : \{invalid\}]$$

Address-of operator - when the address-of (&) operator is used, it creates a temporary variable that points to the operand of &. Let i be $\&v$. Then,

$$\varphi_i(d) = \begin{cases} [d|tmp : \{v\}] & \text{if } v \text{ is a local variable} \\ [d|tmp : \{global\}] & \text{if } v \text{ is a global variable} \end{cases}$$

`Pointer` **dereferencing** - when we dereference a `Pointer`, a temporary variable may be created if the result is not a value. This is useful for cases when `Pointer` to `Pointer` types are used. Let i be $*v$. Then,

$$\varphi_i(d) = \begin{cases} d & \text{if } *v \text{ is a } \texttt{Value} \\ [d|tmp : d(d(v))\}] & \text{otherwise} \end{cases}$$

Memory allocation - when memory is allocated, a new `Owner` is created to represent the memory zone returned by the allocate statement. Let i be `allocate`(x). Then,

$$\varphi_i(d) = [d|\text{alloc}_x : \text{alloc}'_x]$$

Memory deallocation - when memory is deallocated through a pointer p, all owners in its pset are invalidated. Let i be `deallocate`(p). Then,

$$\varphi_i(d) = \text{invalid}_{d(p)}(d)$$

Function calls - when analyzing function calls, we assume that the annotated postconditions are true, and use them as the pset of the function output. Let $i = p = f()$ and the postcondition $post(f,S)$. Then,

$$\varphi_i(d) = [d|v : S]$$

Function definitions - when analyzing function bodies, we assume that the annotated preconditions are true, and use them as the psets of the function parameters. Let $f(x)$ be a function annotated with $pre(x,S)$, Then,

$$\varphi_i(d) = [d|x : S]$$

All other instructions - all other instructions that do not affect memory, there are no changes to the pmap. So for all other i,

$$\varphi_i(d) = d$$

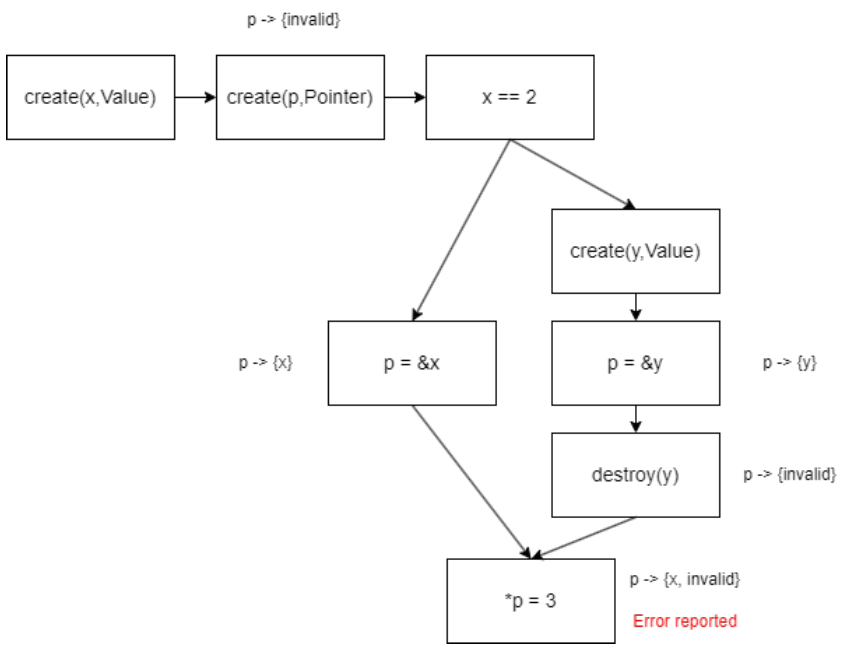


Figure 2: Example CFG analysis

After analyzing the whole CFG and calculating the fixpoint solution of the dataflow system, we have access to the *pmap* of all nodes and can enforce the rules presented above in the following way:

- It is an error to dereference an invalid pointer. If i is $*p$ and `invalid` $\in d(p)$, then we report the error.
- It is an error to dereference an `Owner` that was moved from. If i is $*o$ and `invalid` $\in d(o)$, then we report the error.
- It is an error to pass a `Pointer` as a function argument if it is invalid, or violates the preconditions of the function. Let f be a function annotated with $pre(p,S)$. At every function call $i = f(p)$, if `invalid` $\in d(p)$, an error will be reported. In addition, if there is no $p' \in S$ such that $d(p) = d(p')$, then the precondition is violated and an error will be reported.
- It is an error to return a `Pointer` from a function that is invalid or violates the postconditions of the function. Let f be a function annotated with $post(f,S)$. At every statement that returns, $i =$ return p, if `invalid` $\in d(p)$, an error will be reported. In addition, if there is no $p' \in S$ such that $d(P) = d(p')$, then the postcondition is violated and an error will be reported

Figure 2 shows an example analysis of the program in listing 4 annotated with the relevant changes in the *pmap* entry of p.

4 Implementation and Evaluation

4.1 Implementation

The implementation of the framework consists of three main components:

- Parsing and processing a C++ source file received as input in order to transform it into the programming language used during the analysis.
- Creating the control flow graph and solving the dataflow system in order to calculate the *pmap* at each node.
- Using the *pmap*s obtained after the previous step, identify and report any errors.

In addition to this, additional work has been done to improve error messages and help identify the source of errors faster. The main feature created with this purpose is the ability to track the moments when pointers are invalidated, and include these locations when an error is reported.

Because the main interest of our framework is not parsing the language, but performing static analysis on it, choosing an already implemented solution for this would be the best choice. Not only does this save resources because we do not need to implement a parser from scratch, but also it provides a well-tested, maintained, and well-documented parser which we can take advantage of.

The tool we chose for parsing the source files is Clang [2]. Clang is an open-source project that provides a language front-end and tooling infrastructure for languages in the C family. It is considered one of the three main C++ compilers and is widely used to build production-quality applications [8].

4.2 Evaluation

During each stage of development of the framework, several tests were run to evaluate the performance and accuracy of each component. Both handwritten, specific tests for certain functionalities, and real-world snippets and projects have been used to be able to accurately evaluate the framework.

Type classification tests

Being one of the first steps of the analysis, type classification plays an important role and any misclassification will influence all further steps in the analysis, possibly leading to erroneous reporting of bugs. This is why we considered it necessary during development to keep a 100% pass rate of tests in this category, as any errors here would make the evaluation of other features impossible. This also meant that the quality of the test had to be high, covering as many cases as possible

The test set consists of 30 tests and was created to cover every rule specified in Section 3.2.1, using multiple approaches. Both user-created and standard library types were used in the tests, in as many combinations as possible.

The framework passes all 30 tests, and the distribution of tests is presented in Table 1.

Table 1: Type classification tests

Type classification 1	Total tests	User types	Std types	Both types	Pass rate
`Owner`	14	6	5	3	100%
`Pointer`	10	3	5	2	100%
`Value`	6	3	2	1	100%

Error detection tests

To evaluate the accuracy of the framework, we categorized the results of each error detection test as true positive, false positive, true negative, and false negative.

The most important metric to take into consideration is the number of false negatives. Our goal is to minimize this number, so that as few errors are missed as possible. The next priority is minimizing the number of false positives, as flooding the user with reports of inexistent errors might make them miss a true positive report.

The test set constructed for this evaluation aims to cover both erroneous as well as error-free code. The errors present should follow as many different patterns as possible. One important aspect is that different people write code in different manners, depending on their skills, experience, influences, and time invested. We included three different styles of writing code to try to emulate real-world codebases:

Advanced - this style of code is compliant with modern C++ best practices (see for example [1]), heavily uses the standard library, and designs clean and correct code. This would correspond to code written by a very experienced programmer who rarely makes mistakes.

Regular - this style of code uses only the most well-known parts of the standard library while implementing other algorithms and data structures from scratch. This style would be the most common in real-world projects, corresponding to an experienced programmer, that knows and applies most best practices.

Basic - this style of code barely uses the standard library, while possibly using C++ anti-patterns that may cause bugs in some situations. This style would be common among inexperienced programmers, or old codebases that have not been modernized.

The test suite consists of 102 tests and the results of the evaluation can be seen in Table 2.

Table 2: Error detection tests

Code style	Total tests	False Negative	False Positive	Accuracy
Advanced	33	0	1	96.96%
Regular	35	3	2	85.7%
Basic	34	6	10	52.94%

From the results, it can be concluded that the accuracy of error detection increases the more modern the code is. One cause of this is that by aligning with modern standards and writing code with best practices in mind, like const-correctness, using the standard library when possible, or designing classes with intuitive interfaces, the static analyzer can infer much more information about the context of the program. While the programmer can deduce most information from context, patterns or naming conventions, the compiler has no way doing this. Some examples of failed tests and potential solutions are presented below.

Analyzing Real-World Code

In addition to evaluating the framework on small snippets of code, it is important to test its behavior in a real-world use case, on code bases of different sizes. We have chosen 5 projects of sizes from a few hundred lines to over 100000, and ran the analysis of them, checking the number of errors and validating whether they were correctly identified.

Project A is a small application written as a helper tool. It contains approximately 500 lines of modern C++ and implements simple functionalities like reading and writing files on the disk and manipulating strings.

Project B is a project that is currently in development and has around 4000 lines of code. It implements medium-complexity functionalities for filtering and grouping data from multiple sources.

Project C is a production application that is used as an interpreter for a domain-specific language. It is a medium-size code base of around 15 thousand lines of modern C++.

Project D is a complex application, containing over 40 thousand lines of code, written over a few years. Its features are of high complexity and include parsing data streams, extracting and correlating information, and using it to generate reports.

Project E is a very old code base, comprised of over 100 thousand lines of mostly C and C++98. It contains multiple modules, with very complex features and multiple interactions with operating system interfaces.

The results and the evaluation can be found in Table 3 and Table 4.

Table 3: Error detection tests

Project	Coding style	Errors	False Positives	FP rate
A	Advanced	0	0	0%
B	Regular	6	5	83.33%
C	Advanced	4	1	25%
D	Regular	105	105	100%
E	Basic	1038	??	??%

Table 4: Compilation speed tests

Project	Compilation	Compilation + Analysis	Delta	Slowdown
A	53 ms	68 ms	13 ms	24.5%
B	6272 ms	9057 ms	2785 ms	44.4%
C	54 s	76 s	22 s	40.7%
D	106 s	139 s	33 s	31.1%
E	583 s	721 s	139 s	23.8%

It can be seen that the more modern and well-maintained the code is, the better the framework behaves. Although project C has almost 4 times as many lines of code as project B, the false positive rate is much lower. This may be attributed to the more modern design of the code and the much heavier use of standard library functionalities that enable the analysis to accurately infer the needed information.

The results of project D also bring up a potentially important issue: all 105 reported errors were false positives, having the same root cause: not annotating functions and variables as const when needed.

Project E generated so many errors caused by a variety of anti-patterns, legacy code, and lack of refactorization when needed, that it was impossible to manually validate each one. This may raise a problem when the framework has to analyze huge, legacy code bases: flooding the user with errors, most of which are false positives, caused mostly by legacy code, which would require a significant time investment to solve.

Discussion

The evaluation of the framework showed good results, but at the same time, the false positives and false negatives show that there are more improvements to be made. The main causes of test failures we have been able to identify are:

- User-defined types that behave like an Owner, but don't fully follow the specification in our framework.

- Types that have shared-ownership behaviour (for example $std :: shared_ptr$.
- Functions that have no side-effects, but are not annotated $const$.

Further improvements of type classification rules and program analysis are needed to reduce these failures.

5 Conclusions and Future Work

Conclusions The goal when the development of the framework started was to be able to prevent some exploits from ever becoming available to the public by detecting the errors that would cause the vulnerability during the development process, and warning the programmer so that it can be fixed.

The framework we developed is able to take any C++ source file as input, using even the most recently standardized features, facilitated by Clang's parser and AST. Then, it will transform the AST into a control flow graph, adapting the original program to the language used during the analysis, by transforming its types, instructions and functions. After the creation of the CFG, a dataflow system is defined and solved for each function in the program, to extract the necessary information about the points-to maps at each node. Finally, using this information, a set of rules is enforced, and errors are reported to the programmer through intuitive messages, that contain the location of the bug, in addition to potential previous instructions that may cause it.

Multiple tests were performed in order to evaluate the accuracy and speed of the framework. While most tests were successful, some of them showed some potential problems when analyzing big codebases. The main one is that for human programmers, it may be easy to understand information from the context of the code, without the need of explicit annotations, while for static analyzers that is impossible.

It can be concluded that our framework achieved its goals, as it is able to detect multiple patterns of use-after-free bugs, and correctly report the errors to the programmer.

Future Work While we can say that the framework was able to achieve its goals, there are many improvements that can be made. Some of the most important are:

- Modifying the type classification rules to be able to recognise shared ownership semantics. Some cases of false positives are caused by types that have the behavior of a shared `Owner`, which invalidate all Pointers when a single `Owner` is destroyed, while in reality the Pointers remain valid.
- Improve type classification to be able to identify types which represent Owners, but don't fall into any of the categories mentioned in our rules.
- Implement mechanisms that are able to deduce some information without it being explicitly mentioned. One good example is being able to identify whether a variable or function can be considered const.
- Research interprocedural analysis to be able to detect much more patterns of errors.

References

[1] *C++ Core Guidelines.* Available at `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`. `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines` last visited in June 2024.

[2] *Clang ecosystem*. Available at `https://clang.llvm.org/`. `https://clang.llvm.org/` last visited in June 2024.

[3] *Clang static analyzer checkers*. Available at `https://clang.llvm.org/docs/analyzer/checkers.html`. `https://clang.llvm.org/docs/analyzer/checkers.html` last visited in June 2024.

[4] *CLion warnings*. Available at `https://www.jetbrains.com/help/clion/list-of-c-cpp-inspections.html`. `https://www.jetbrains.com/help/clion/list-of-c-cpp-inspections.html` last visited in June 2024.

[5] *The Bluekeep bug*. Available at `https://en.wikipedia.org/wiki/BlueKeep`. `https://en.wikipedia.org/wiki/BlueKeep` last visited in June 2024.

[6] *The EternalBlue bug*. Available at `https://en.wikipedia.org/wiki/EternalBlue`. `https://en.wikipedia.org/wiki/EternalBlue` last visited in June 2024.

[7] *The Heartbleed bug*. Available at `https://heartbleed.com/`. `https://heartbleed.com/` last visited in June 2024.

[8] *The State of Developer Ecosystem 2023 - Jetbrains*. Available at `https://www.jetbrains.com/lp/devecosystem-2023/cpp/`. `https://www.jetbrains.com/lp/devecosystem-2023/cpp/` last visited in June 2024.

[9] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt & Marc Heuse (2020): *AFL++ : Combining Incremental Steps of Fuzzing Research*. In Yuval Yarom & Sarah Zennou, editors: *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, USENIX Association, doi:10.5555/3488877.3488887. Available at `https://www.usenix.org/conference/woot20/presentation/fioraldi`.

[10] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs & Koen Langendoen (2002): *Modern Compiler Design*. John Wiley.

[11] Herb Sutter: *Lifetime safety: Preventing common dangling*. Available at `https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf`. `https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf` last visited in June 2024.

[12] International Organization for Standardization (2020): *Programming Languages - C++*, 5th edition. ISO/IEC 14882:2020, International Organization for Standardization, Geneva, Switzerland.

[13] Csaba Krasznay (2020): *Case Study: The NotPetya Campaign*, pp. 485–499.

[14] M. Satheesh Kumar, Jalel Ben-Othman & K. G. Srinivasagan (2018): *An Investigation on Wannacry Ransomware and its Detection*. In: *2018 IEEE Symposium on Computers and Communications, ISCC 2018, Natal, Brazil, June 25-28, 2018*, IEEE, pp. 1–6, doi:10.1109/ISCC.2018.8538354.

[15] John R. Levine (2009): *flex and bison - Unix text processing tools*. O'Reilly. Available at `http://www.oreilly.de/catalog/9780596155971/index.html`.

[16] Nicholas Nethercote & Julian Seward (2007): *Valgrind: a framework for heavyweight dynamic binary instrumentation*. In Jeanne Ferrante & Kathryn S. McKinley, editors: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, ACM, pp. 89–100, doi:10.1145/1250734.1250746.

[17] Flemming Nielson, Hanne Riis Nielson & Chris Hankin (1999): *Principles of program analysis*. Springer, doi:10.1007/978-3-662-03811-6.

[18] Konstantin Serebryany, Derek Bruening, Alexander Potapenko & Dmitriy Vyukov (2012): *AddressSanitizer: A Fast Address Sanity Checker*. In Gernot Heiser & Wilson C. Hsieh, editors: *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, USENIX Association, pp. 309–318, doi:10.5555/2342821.2342849. Available at `https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany`.

[19] Kosta Serebryany (2016): *Continuous Fuzzing with libFuzzer and AddressSanitizer*. In: *2016 IEEE Cybersecurity Development (SecDev)*, pp. 157–157, doi:10.1109/SecDev.2016.043.

[20] W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim & Michael F. Siok (2010): *Recent Catastrophic Accidents: Investigating How Software was Responsible*. In: *Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010*, IEEE Computer Society, pp. 14–22, doi:10.1109/SSIRI.2010.38.

A Type System for Data Flow and Alias Analysis in ReScript

Nicky Ask Lund
Department of Computer Science
Aalborg University, Denmark
loevendallund@gmail.com

Hans Hüttel
Department of Computer Science
University of Copenhagen, Denmark
hans.huttel@di.ku.dk

ReScript is a strongly typed language that targets JavaScript, as an alternative to gradually typed languages, such as TypeScript. In this paper, we present a sound type system for data-flow analysis for a subset of the ReScript language, more specifically for a λ-calculus with mutability and pattern matching. The type system is a local analysis that collects information about variables that are used at each program point as well as alias information.

1 Introduction

The goal of data-flow analysis is to provide a static analysis of the flow information in a program that can be used in compiler optimizations and for register allocation. The original approach is to build a system of flow equations based on a graph representation of the program and to compute a solution using an iterative algorithm [3, 8]. Other graph-free approaches have also been considered [6]. A challenge in this setting is how to deal with the aliasing that imperative language constructs introduce.

Type systems have often been used to provide static analyses of programs in order to characterize specific run-time errors, including ones caused by aliasing. In [9] Smith et al. present a notion of alias types that allows functions to specify the shape of a store and to track the flow of pointers through a computation. The language is a simple location-based language. Other type systems are substructural. Ahmed et al. [7] use a language based on a linear λ-calculus to give an alternative formulation of alias types. The type system crucially relies on linearity, and every well-typed program terminates.

In this paper we present a type system for data flow analysis in the presence of aliasing for a non-trivial fragment of the programming language ReScript which is meant as an alternative to other typed languages that target JavaScript. ReScript is based on OCaml with a JavaScript-inspired syntax and a type system based on that of OCaml[1]. ReScript is imperative and allows for mutability through reference constructs for creation, reading, and writing.

The fragment that we study incorporates both functional and imperative features. We show that the type system is sound in the sense that it correctly overapproximates the set of occurrences on which any given occurrence depends. Moreover, the dependency information that the type system provides can also be used to reason about information flow properties such as non-interference. Furthermore, an implementation for the type system has been made to demonstrate the type system. The full version of our paper is available at [5].

2 A fragment of ReScript

In this paper we consider a fragment of ReScript that contains a λ-calculus with pattern matching, local declarations that can be recursive and a notion of mutable references.

2.1 Syntax

In a data-flow analysis we must record information of where variables are used. Therefore, the language presented is extended with a notion of *program points* taken from a countably infinite set $\mathbf{P}$. Every subexpression is labelled with a unique program point. *Occurrences* $o \in \mathbf{Occ}$ are labelled expressions e^p where $e \in \mathbf{Exp}$ and $p \in \mathbf{P}$. We let ℓ range over a countably infinite, totally ordered set of locations $\mathbf{Loc}$ and x, f range over the set of variables $\mathbf{Var}$. An occurrence is *atomic* if it is of the form u^p where $u \in \mathbf{Var} \cup \mathbf{Loc}$. If u^p is atomic, we call any other occurrence u^q a *u-occurrence*.

When given a syntactic category $\mathbf{C}$, we let $\mathbf{C_P}$ denote the pair $\mathbf{C} \times \mathbf{P}$, so that e.g. $\mathbf{Exp_P} = \mathbf{Exp} \times \mathbf{P}$. This means that $\mathbf{Occ} = \mathbf{Exp_P}$.

The formation rules of our abstract syntax are shown below.

$$\begin{aligned}
&o ::= e^p\\
&e ::= x \mid c \mid o_1\ o_2 \mid \lambda x.o \mid c\ o_1\ o_2\\
&\quad \mid \text{let } f\ o_1\ o_2 \mid \text{let rec } f\ o_1\ o_2 \mid \text{case } o_1\ \vec{\pi}\ \vec{o} \mid \text{ref } o \mid o_1 := o_2 \mid !o\\
&\pi ::= n \mid b \mid x \mid \_ \mid (s_1, \cdots, s_n)
\end{aligned}$$

An *abstraction* $\lambda\ x.o$ has a parameter x and body o. *Constants* c are either natural numbers n, boolean values b, unit value $()$, or functional constants, such as the arithmetic operations and the Boolean connectives.

An *application* is written $o_1\ o_2$, and $c\ o_1\ o_2$ denotes a *functional application* where c is a functional constant and o_1, o_2 are its arguments.

Local declarations let $f\ o_1\ o_2$ associate the variable f with the value o_1 within o_2, and let rec $f\ o_1\ o_2$ allows us to define a recursive function f where f may occur in o_1. ReScript is an imperative language due to the presence of the *reference* construct ref o which creates a reference in the form of a location and allows for binding locations to local declarations. We can read from a reference o by writing $!o$ and write to a reference using the *assignment* construct $o_1\ :=\ o_2$.

The pattern matching construct case $o_1\ \vec{\pi}\ \vec{o}$ matches an occurrence with the ordered set, $\vec{\pi}$, of patterns. We denote the size of the tuple pattern π by $|\vec{\pi}|$ and the size of a tuple occurrence by $|\vec{o}|$, requiring that $|\vec{\pi}| = |\vec{o}|$ such that for each pattern in $\vec{\pi}$ there is a clause in $\vec{o}$.

The notions of free and bound variables are defined as expected. We assume that all binding occurrences involve distinct names; this can be ensured by means of α-conversion.

Example 1. Consider

```
(let x (ref 484000¹)²
  (let y (let z (5³)⁴
      (x⁵:=z⁷)⁸)⁹  (!x)¹⁰)¹¹)¹²
```

This creates a reference to the constant 3 and binds the reference to x (so x is an alias of this reference). Next a binding of z is made to the constant 5 before writing to the reference, that x is bound to, to the value that z is bound to. Then a binding for y is made to the unit value, as the assignment evaluates to the unit value. Lastly the reference, that x is bound to, is read.

2.2 The binding model

Our binding model uses an environment *env* that keeps track of the bindings of variables to values. Values are given by the formation rules

$$v ::= c \mid \ell \mid () \mid \langle x, e^{p'}, env\rangle \mid \langle x, f, e^{p''}, env\rangle$$

Constants c, locations ℓ, and unit $()$ are values, as are closures, $\langle x, e^{p'}, env\rangle$ and recursive closures, $\langle x, f, e^{p''}, env\rangle$.

An environment $env \in \mathbf{Env}$ is a partial function $env : \mathbf{Var} \rightharpoonup \mathbf{Values}$ and we let $env^{-1}(v) = \{x \in \mathrm{dom}(env) \mid env(x) = v\}$. A store $sto \in \mathbf{Sto}$ is a partial function $\mathbf{Sto} = \mathbf{Loc} \cup \{next\} \rightharpoonup \mathbf{Values}$ where $next$ is a pointer to the next unused location – this information is needed when new locations are needed.

Moreover, we assume a function $new : \mathbf{Loc} \to \mathbf{Loc}$, which given, a location, gives us the next location.

For any function f we let $f[u \mapsto w]$ denote the function f' such that $f(u') = f(u)$ for $u \neq u'$ and $f'(u') = w$.

2.3 Keeping track of dependencies

The semantics that follows will collect the semantic dependencies in a computation. An occurrence u^p semantically depends upon a set of occurrences S if the value of u^p can be found using at most the values of the occurrences in S. To determine semantic dependencies we use a dependency function that will tell us for each variable and location occurrence what other, previous occurrences they depend upon.

Definition 1 (Dependency function)**.** A dependency function w is a partial functions from atomic occurrences to a pair of dependencies:

$$w : \mathbf{Loc_P} \cup \mathbf{Var_P} \rightharpoonup \mathbb{P}(\mathbf{Loc_P}) \times \mathbb{P}(\mathbf{Var_P})$$

For a dependency function w and a $u^p \in \mathbf{Loc_P} \cup \mathbf{Var_P}$, the clause

$$w(u^p) = (L, V)$$

tells us that the element u^p is bound to a pair of location and variable occurrences where L is a set of location occurrences $L = \{\ell_1^{p_1}, \cdots, \ell_n^{p_n}\}$ and a set of variable occurrences $V = \{x_1^{p'_1}, \cdots, x_m^{p'_m}\}$, meaning that the value of the element u^p depends on the occurrences found in L and in V.

Example 2. Consider the occurrence from Example 1, where we can infer the following bindings for a dependency function w_{ex} over this occurrence:

$$w_{ex} = \begin{array}{l}[x^2 \mapsto (\emptyset,\emptyset), z^4 \mapsto (\emptyset,\emptyset), y^9 \mapsto (\emptyset, \{x^5\}),\\ \ell^2 \mapsto (\emptyset,\emptyset), \ell^8 \mapsto (\emptyset, \{z^7\})]\end{array}$$

where ℓ is the location created from the reference construct. The variable bindings are distinct, as the location ℓ is bound multiple times, for the program points 2 and 8.

When we read an occurrence bound in w_{ex}, we must also know its program point, as there can exists multiple bindings for the same variable or location.

By considering Example 2, we would like to read the information from the location, that x is an alias to. As it is visible from the occurrence in Example 1, we know that we should read from ℓ^8, since we wrote to that reference at the program point 8. From w_{ex} alone it is not possible to know which occurrence to read, since there is no order defined between the bindings. We therefore introduce a notion of ordering in the form of a binary relation over program points.

Definition 2 (Occurring program points)**.** Let O be a set of occurrences, then $\mathsf{points}(O)$ is given by:

$$\mathsf{points}(O) = \{p \in \mathbf{P} \mid \exists e^p . e^p \in O\}$$

For a pair (L, V) we let $\mathsf{points}(L, V) = \mathsf{points}(L) \cup \mathsf{points}(V)$.

Any dependency function induces an ordering on program points as follows.

Definition 3. Let $w \in \mathbf{W}$ be a dependency function. Then the induced order $\sqsubset_w$ is given by

$$\sqsubset_w = \{(p, p') \mid p \in \mathsf{points}(\mathrm{dom}(w)), p' \in \mathsf{points}(w(p))\}$$

We say that w is a partial order if its equality closure $\sqsubset_w$ is a partial order.

Example 3. Consider the example from Example 2. Assume a binary relation over the dependency function w_{ex} given by

$$\sqsubset_{w_{ex}} = \{(2,4), (2,9), (5,9), (2,8), (7,8)\}$$

From this ordering, it is easy to see the ordering of the elements. The ordering we present also respects the flow the occurrence from Example 1 would evaluate to. We then know that the dependencies for the reference (that x is an alias to) is for the largest binding of ℓ.

The immediate predecessor $\mathsf{IP}(u, S)$ of an element u^p wrt. a set of occurrences S is the most recent u-element in S seen before u.

Definition 4 (Immediate predecessor)**.** Let u be an element, let $\sqsubset$ be an ordering on program points and S be a set of occurrences, then $\mathsf{IP}(u, S)$ is given by

$$\mathsf{IP}(u^p, S) = \sup\{u^q \in S \mid q \sqsubset p\}$$

Based on Definition 4, we can present an instantiation of the function for the dependency function w and an order over w, $\sqsubset_w$:

Definition 5. Let w be a dependency function, $\sqsubset_w$ be the induced order, and u be an element, then $\mathsf{IP}_{\sqsubset_w}$ is given by:

$$\mathsf{IP}_{\sqsubset_w}(u, w) = \sup\{u^p \in \mathrm{dom}(w) \mid u^q \in \mathrm{dom}(w).q \sqsubset_w p\}$$

Example 4. As a continuation of Example 3, we can now find the greatest element for an element, e.g., a variable or location. As we were interested in finding the greatest bindings a location is bound to in $w_e x$, we use the function $\mathsf{IP}_{\sqsubset_w}$:

$$\mathsf{IP}_{\sqsubset_{w_e x}}(\ell, w_e x) = \sup\{\ell^p \in \mathrm{dom}(w) \mid \ell^q \in \mathrm{dom}(w).q \sqsubset_{w_e x} p\}$$

where the set we get for ℓ is $\{\ell^2, \ell^8\}$. From this, we find the greatest element:

$$\ell^8 = \sup\{\ell^2, \ell^8\}$$

As we can see, from the $\mathsf{IP}_{w_e x}$ function, we got ℓ^8 which were the occurrence we wanted.

2.4 Collecting semantics

The semantics for our language that collects dependency information is a big-step semantics with transitions of the form

$$env \vdash \left\langle e^{p'}, sto, (w, \sqsubset_w), p \right\rangle \to \langle v, sto', (w', \sqsubset'_w), (L, V), p'' \rangle$$

This should be read as: Given the store sto, a dependency function w, a relation over w, and the previous program point p, the occurrence $e^{p'}$ evaluates to a value v, an updated store sto', an updated dependency function w', a relation over w', the dependency pair (L,V), and the program point p'' reached after evaluating $e^{p'}$, given the bindings in the environment *env*.

A selection of the rules for $\rightarrow$ can be found in Table 1.

The (VAR) rule uses the environment to get the value x is bound to and uses dependency function w to get its dependencies. To lookup the dependencies, the function $\mathsf{lP}_{\sqsubset_w}$ is used to get the greatest binding a variable is bound to, in respect to the ordering $\sqsubset_w$. Since the occurrence of x is used, it is added to the set of variable occurrences we got from the lookup of the dependencies for x.

The (LET) rule for the occurrence $[\text{let}\, x\, e_1^{p_1}\, e_2^{p_2}]^{p'}$, creates a local binding that can be used in $e_2^{p_2}$. The (LET) rule evaluate $e_1^{p_1}$, to get the value v, that x will be bound to in the environment for $e_2^{p_2}$, and the dependencies used to evaluate $e_1^{p_1}$ are bound in the dependency function. As we reach the program point p_1 after evaluating $e_1^{p_1}$, and it is also the program point before evaluating $e_2^{p_2}$, the binding of x in w is to the program points p_1.

The (REF) rule, for the occurrence $[\text{ref}\, e^{p'}]^{p''}$, creates a new location and binds it in the store *sto*, to the value evaluated from $e^{p'}$. We record the dependencies from evaluating the body $e^{p'}$ in w at the program point p''.

(REF-READ) evaluates the body e^{p_1} to a value which must be a location ℓ, and reads the value of ℓ in the store. The (REF-READ) rule looks up the dependencies for ℓ in w. As there could be multiple bindings for ℓ, in w, at different program points, we use the $\mathsf{lP}_{\sqsubset_{w'}}$ function to get greatest binding of ℓ with respect to the ordering $\sqsubset_{w'}$, and we also add the location occurrence $\ell^{p'}$ to the set of locations.

Finally (REF-WRITE) tells us that we must evaluate $e_1^{p_1}$ to a location ℓ and $e_2^{p_2}$ to a value v, and bind ℓ in the store *sto* to the value v. We pass the program point p' and the dependency function is also updated with a new binding for ℓ.

3 A type system for data-flow analysis

The type system for data-flow analysis that we now present is an overapproximation of the big-step semantics.

3.1 An overview of the type system

The system assigns types, presented in Section 3.2, to occurrences given a type environment (presented in Section 3.4) and a so-called basis (presented in Section 3.3).

As presented, the language contains local information as bindings and global information as locations. Since locations are a semantic notion, and references do not need to be bound to variables, we use the notion of *internal* variables to represent locations. Internal variables are denoted by $\nu x, \nu y, \nu z \ldots \in$ **IVar**. We use a partition of **IVar** $\cup$ **Var** to represent aliasing. Whenever variables or internal variables belong to the same subset in a partition, the intention is that they share the same location.

In this paper, we will not introduce polymorphism into the type system. For this reason we require that references cannot be bound to abstractions and that every abstraction is used at most used once.

3.2 Types

The set of types **Types** is defined by the formation rules

$$T ::= (\delta, \kappa) \mid T_1 \rightarrow T_2$$

(VAR)

$$\frac{}{env \vdash \left\langle x^{p'}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle v, sto, (w, \sqsubset_w), (L, V \cup \{x^{p'}\}), p' \right\rangle}$$

where $env(x) = v$, $x^{p''} = \mathsf{IP}_{\sqsubset_w}(x, w)$, and $w(x^{p''}) = (L, V)$

(LET)

$$\frac{\begin{array}{c} env \vdash \left\langle e_1^{p_1}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle v_1, sto_1, (w_1, \sqsubset_w^1), (L_1, V_1), p_1 \right\rangle \\ env[x \mapsto v_1] \vdash \left\langle e_2^{p_2}, sto_1, (w_2, \sqsubset_w^1), p_1 \right\rangle \to \left\langle v, sto', (w', \sqsubset_w'), (L, V), p_2 \right\rangle \end{array}}{env \vdash \left\langle \left[\text{let } x\ e_1^{p_1}\ e_2^{p_2}\right]^{p'}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle v, sto', (w', \sqsubset_w'), (L, V), p' \right\rangle}$$

where $w_2 = w_1[x^{p_1} \mapsto (L, V)]$

(REF)

$$\frac{env \vdash \left\langle e^{p'}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle v, sto', (w', \sqsubset_w'), (L, V), p' \right\rangle}{env \vdash \left\langle \left[\text{ref } e^{p'}\right]^{p''}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle \ell, sto'', (w'', \sqsubset_w'), (\emptyset, \emptyset), p'' \right\rangle}$$

where $\ell = next$, $sto'' = sto'[next \mapsto new(\ell), \ell \mapsto v]$, and
$w'' = w'[\ell^{p'} \mapsto (L, V)]$

(REF-READ)

$$\frac{env \vdash \left\langle e^{p_1}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle \ell, sto', (w', \sqsubset_w'), (L_1, V_1), p_1 \right\rangle}{env \vdash \left\langle [!e^{p_1}]^{p'}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle v, sto', (w', \sqsubset_w'), (L \cup L_1 \cup \{\ell^{p''}\}, V \cup V_1), p' \right\rangle}$$

where $sto'(\ell) = v$, $\ell^{p''} = \mathsf{IP}_{\sqsubset_w'}(\ell, w')$, and $w'(\ell^{p''}) = (L, V)$

(REF-WRITE)

$$\frac{\begin{array}{c} env \vdash \left\langle e_1^{p_1}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle \ell, sto_1, (w_1, \sqsubset_w^1), (L_1, V_1), p_1 \right\rangle \\ env \vdash \left\langle e_2^{p_2}, sto_1, (w_1, \sqsubset_w^1), p_1 \right\rangle \to \left\langle v, sto_2, (w_2, \sqsubset_w^2), (L_2, V_2), p_2 \right\rangle \end{array}}{env \vdash \left\langle \left[e_1^{p_1} := e_2^{p_2}\right]^{p'}, sto, (w, \sqsubset_w), p \right\rangle \to \left\langle (), sto', (w', \sqsubset_w'), (L_1, V_1), p' \right\rangle}$$

where $sto' = sto_2[\ell \mapsto v]$, $\ell^{p'} = inf_{\sqsubset_w^2}\ \ell, w$,
$w' = w_2[\ell^{p'} \mapsto (L_2, V_2)]$, and $\sqsubset_w' = \sqsubset_w^2 \cup (p'', p')$

Table 1: Selected rules from the semantics

If an occurrence o has the base type (δ, κ), the set δ is the set of occurrences that the value of o can depend on, and κ represents alias information in the form of the set of variables and internal variables upon which the value of o may depend. If o has type $\kappa \neq \emptyset$, the occurrence must therefore represent a location.

Definition 6 (Type base for aliasing)**.** For an occurrence o, let $\mathbf{Var}_o$ be the set of all variables found in o and $\mathbf{IVar}_o$ be the set of all internal variables found in o. The type base $\kappa^0 = \{\kappa_1^0, \cdots, \kappa_n^0\}$ is then a partition of $\mathbf{Var}_o \cup \mathbf{IVar}_o$, where $\kappa_i^0 \cap \kappa_j^0 = \emptyset$ for all $i \neq j$.

For a variable, x to be an alias of an internal variable, νy, there must exist a κ_i^0 where $x \in \kappa_i^0$ and $\nu y \in \kappa_i^0$. This means that there can only be more than one variable in a κ_i^0, if there also exists an internal variable in κ_i^0.

The arrow type is introduced to type abstractions. If either T_1 or T_2 in an arrow type $T_1 \to T_2$ is a base type where $\kappa \neq \emptyset$, then the abstraction must either take a reference as input or return a reference.

Since the type system approximates the occurrences used to evaluate an occurrence, we need a notion of combining types.

Definition 7. Let T_1 and T_2 be two types, then their union is defined as

$$T_1 \cup T_2 = \begin{cases} (\delta \cup \delta', \kappa \cup \kappa') & \begin{array}{l} T_1 = (\delta, \kappa) \\ T_2 = (\delta', \kappa') \end{array} \\ (T_1' \cup T_2') \to (T_1'' \cup T_2'') & \begin{array}{l} T_1 = T_1' \to T_1'' \\ ; T_2 = T_2' \to T_2'' \end{array} \\ (\delta' \cup \delta, \kappa \cup \kappa') & \begin{array}{l} T_1 = (\delta', \kappa') \\ T_2 = (\delta, \kappa) \end{array} \\ \text{undefined} & \text{otherwise} \end{cases}$$

3.3 The binding model of the type system

The semantics will let us find dependency information, and the type system must approximate these semantic notions.

A type environment tells us the types of elements.

Definition 8 (Type Environment)**.** A type environment Γ is a partial function $\Gamma : \mathbf{Var_P} \cup \mathbf{IVar_P} \rightharpoonup \mathbf{Types}$

Definition 9 (Updating a type environment)**.** Let Γ be a type environment, let u^p be an element and let T be a type. We write $\Gamma[u^p : T]$ to denote the type environment Γ' where:

$$\Gamma'(y^{p'}) = \begin{cases} \Gamma(y^{p'}) & \text{if } y^{p'} \neq u^p \\ T & \text{if } y^{p'} = u^p \end{cases}$$

We also assume an ordering of program points at type level.

Definition 10 (Approximated order of program points)**.** An approximated order of program points Π is a pair

$$\Pi = (\mathbf{P}, \sqsubset_\Pi)$$

where

- $\mathbf{P}$ is the set of program points in an occurrence,

- $\sqsubset_\Pi \subseteq \mathbf{P} \times \mathbf{P}$

We say that Π is a partial order if $\sqsubset_\Pi$ is a partial order.

The notion of the immediate predecessor of a $u \in \mathbf{IVar} \cup \mathbf{Var}$ at the type level is relative to a type environment Γ and the approximated order Π.

Definition 11 (Immediate predecessor at type level)**.**

$$\mathsf{IP}_{\sqsubset_\Pi}(u,\Gamma) = \sup\{u^p \in \mathrm{dom}(\Gamma) \mid u^q \in \mathrm{dom}(\Gamma).q \sqsubset_\Pi p\}$$

A lookup of variable u^p in the semantics is straightforward as its value will be unique. In the type system, however, we need to approximate over all possible branches in an occurrence. To this end, we consider chains wrt. our approximate order. A p-chain describes the history, or a single possible evalution, behind an occurrence u^p. A set of p-chains can thus be used to describe what an internal variable depends on.

Definition 12 (p-chains)**.** A p-chain, denoted as Π^*_p, is a maximal chain wrt. $\sqsubset_\Pi$ whose maximal element is p. We write $\Pi^*_p \in \Pi$, if Π^*_p is a Π-chain. For any p, we let Υ_p denote the set of all p-chains in Π.

We can now define the immediate predecesor for the type system wrt. the set of p-chains. This is done by taking the union of all p-chains for an occurrence u^p.

Definition 13. Let $u \in \mathbf{Var} \cup \mathbf{IVar}$, be either a variable or internal variable, Γ be a type environment, and Υ_p be a set of p-chains, then IP_{Υ_p} is given by:

$$\mathsf{IP}_{\Upsilon_p}(u,\Gamma) = \bigcup_{\Pi^*_p \in \Upsilon_p} \mathsf{IP}_{\Pi^*_p}(u,\Gamma)$$

3.4 The type system

We will now present the judgement and type rules for the language, that is, how we assign types to occurrences.

Type judgements have the format

$$\Gamma, \Pi \vdash e^p : T$$

and should be read as: the occurrence e^p has type T, given the dependency bindings Γ and the approximated order of program points Π.

A highlight of type rules can be found in Table 2.

(T-VAR) rule, for occurrence x^p, looks up the type for x in the type environment, by finding the greatest binding using Definition 11, and adding the occurrence x^p to the type.

(T-LET-1) rule, for occurrence $[\mathrm{let}\, x\; e_1^{p_1}\; e_2^{p_2}]^p$, creates a local binding for an internal variable, with the type of $e_1^{p_1}$ that can be used in $e_2^{p_2}$. As such, the rule assumes that the type of $e_1^{p_1}$ is a base type with alias information, i.e., $\kappa \neq \emptyset$. The other cases, when $e_1^{p_1}$ is not a base type with alias information, are handled by the (T-LET-2) rule.

(T-CASE) rule, for occurrence $[\mathrm{case}\, e^p\; \vec{\pi}\; \vec{o}]^{p'}$, is an over-approximation of all cases in the pattern matching expression, by taking an union of the type of each case. Since the type of e^p is used to evaluate the pattern matching, we also add this type to the type of the pattern matching.

(T-REF-READ) rule, for occurrence $[!e^p]^{p'}$, is used to retrieve the type of references, where e^p must be a base type with alias information. Since the language contains pattern matching, there can be multiple internal variables in κ and multiple occurrences to read from. To do the lookup, we use $\mathsf{IP}_{\Upsilon_{p'}}$ to look up all the p'-chains.

(T-VAR)

$$\frac{}{\Gamma,\Pi \vdash x^p : T \sqcup (\{x^p\},\emptyset)}$$

where $x^{p'} = uf_{\sqsubset_\Pi}(x,\Gamma)$, and $\Gamma(x^{p'}) = T$

(T-LET-1)

$$\frac{\begin{array}{c}\Gamma,\Pi \vdash e_1^{p_1} : (\delta,\kappa)\\ \Gamma',\Pi \vdash e_2^{p_2} : T_2\end{array}}{\Gamma,\Pi \vdash [\text{let}\, x\, e_1^{p_1}\, e_2^{p_2}]^p : T_2}$$

where $\Gamma' = \Gamma[x^p : (\delta,\kappa \cup \{x\})]$ and $\kappa \neq \emptyset$

(T-CASE)

$$\frac{\begin{array}{c}\Gamma,\Pi \vdash e^p : (\delta,\kappa)\\ \Gamma',\Pi \vdash e_i^{p_i} : T_i \quad (1 \leq i \leq |\vec{\pi}|)\end{array}}{\Gamma,\Pi \vdash [\text{case}\, e^p\, \vec{\pi}\, \vec{o}]^{p'} : T \sqcup (\delta,\kappa)}$$

where $e_i^{p_i} \in \vec{o}$ and $s_i \in \vec{\pi}$ $T = \bigcup_{i=1}^{|\vec{\pi}|} T_i$, and
$\Gamma' = \Gamma[x^p : (\delta,\kappa)]$ if $s_i = x$

(T-REF-READ)

$$\frac{\Gamma,\Pi \vdash e^p : (\delta,\kappa)}{\Gamma,\Pi \vdash [!e^p]^{p'} : T \cup (\delta \cup \delta',\emptyset)}$$

$$\text{where} \left\{ \begin{array}{l} \kappa \neq \emptyset,\ \delta' = \{\nu x^{p'} \mid \nu x \in \kappa\},\ \nu x_1,\cdots,\nu x_n \in \kappa \\ \{\nu x_1^{p_1},\cdots,\nu x_1^{p_m}\} = uf_{\Upsilon_{p'}}(\nu x_1,\Gamma),\cdots, \\ \quad \{\nu x_n^{p'_1},\cdots,\nu x_n^{p'_s}\} = uf_{\Upsilon_{p'}}(\nu x_n,\Gamma) \\ T = \Gamma(\nu x_1^{p_1}) \cup \cdots \cup \Gamma(\nu x_1^{p_m}) \cup \cdots \cup \\ \quad \Gamma(\nu x_n^{p'_1}) \cup \cdots \cup \Gamma(\nu x_n^{p'_s}) \end{array} \right\}$$

Table 2: Selected rules from the type system

4 Soundness

The type system is sound in that the type of an occurrence correspond to the dependencies and the alias information from the semantics. To show this, we will first introduce the type rules for values and then describe relation between the semantics and the type system.

4.1 Type rules for values

In our soundness theorem and its proof, values are mentioned. We therefore state a collection of type rules for the values presented int Section 2.2. The type rules are given in Table 3. We describe the central ones here.

(CONSTANT) differs from the rule (T-CONST), since most occurrences can evaluate to a constant and as such we know that its type should be a base type. Constants can depend on other occurrences; we know that δ can be non-empty, but since constants are not locations, we also know that it cannot contain alias information, and as such κ should be empty.

(LOCATION) types locations, and their type must be a base type. Since locations can depend on other

(CONSTANT)
$$\frac{}{\Gamma,\Pi \vdash c : (\delta, \emptyset)}$$

(LOCATION)
$$\frac{}{\Gamma,\Pi \vdash \ell : (\delta, \kappa)}$$
Where $\kappa \neq \emptyset$

(CLOSURE)
$$\frac{\Gamma,\Pi \vdash env \qquad \Gamma[x^p : T_1],\Pi \vdash e^{p'} : T_2}{\Gamma,\Pi \vdash \left\langle x^p, e^{p'}, env \right\rangle^{p''} : T_1 \to T_2}$$

Table 3: Type rules for values

occurrences, we know that δ can be non-empty. As locations can contains alias information, and that a location is considered to always be an alias to itself, we know that κ can never be empty, as it should always contain an internal variable.

(CLOSURE) type rule represents abstraction, and as such we know that it should have the abstraction type, $T_1 \to T_2$, where we need to make an assumption about the argument type T_1. Since a closure contains the parameter, body, and the environment for an abstraction from when it were declared, we also need to handle those part in the type rule.

The components of the closure are handled in the premises, where the environment must be well-typed. We also type the body of the abstraction in a type environment updated with the type T_1 of its parameter.

As closures and recursive closures contain an environment, we also need to define what it means to be a well-typed environment *env* wrt. a type environment: Every variable bound in *env* is bound to a value that is well-typed wrt. Γ.

Definition 14 (Well-typed environments)**.** Let $v_1, \cdots, v_n$ be values such that $\Gamma,\Pi \vdash v_i : T_i$, for $1 \leq i \leq n$. Let *env* be an environment given by $env = [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$, Γ be a type environment, and Π be the approximated order of program points. We say that:

$$\Gamma,\Pi \vdash env$$

iff

- For all $x_i \in \text{dom}(env)$ then $\exists x_i^p \in \text{dom}(\Gamma)$ where $\Gamma(x_i^p) = T_i$ then

$$\Gamma,\Pi \vdash env(x_i) : T_i$$

4.2 Notions of agreement

As our soundness theorem relates the type system to the semantics, we must define what it means for instances of the binding models of the semantics and the type system to agree.

First we define what it means for a set of occurrences δ to faithfully represent the information from a dependency pair (L,V) wrt. a environment *env*.

Definition 15 (Dependency agreement)**.** We say that:

$$(env,(L,V)) \models \delta$$

if

- $V \subseteq \delta$,
- For all $\ell^p \in L$ where $env^\ell \neq \emptyset$, we then have $env^{-1}\ell \subseteq \kappa_i^0$ for some $\kappa_i^0 \in \delta$
- For all $\ell^p \in L$ where $env^\ell = \emptyset$ then there exists a $\kappa_i^0 \in \delta$ such that $\kappa_i^0 \subseteq$ **IVar**

Since types can contain alias information κ, we also need to define what it means for the information in κ to be known to an environment *env*. If there exists alias information in *env*, then there exists an alias base $\kappa_i^0 \in \kappa^0$ such that the alias information known to *env* is included in that of κ_i^0, and there exists a $\nu x \in \kappa$, such that $\nu x \in \kappa_i^0$. If there is no currently known alias information, we simply check that there exists a corresponding internal variable, that is part of an alias base.

Definition 16 (Alias agreement)**.** We say that

$$(env,(w,\sqsubset_w),\ell) \models (\Gamma,\kappa)$$

if

- $\exists \ell^p \in \mathrm{dom}(w).\nu x^p \in \mathrm{dom}(\Gamma) \Rightarrow \nu x \in \kappa$
- $env^{-1}(\ell) \neq \emptyset.\exists \kappa_i^0 \in \kappa^0 \Rightarrow (env^{-1}(\ell) \subseteq \kappa_i^0) \wedge (\exists \ell^p \in \mathrm{dom}(w).\nu x^p \in \mathrm{dom}(\Gamma) \Rightarrow \nu x \in \kappa_i^0 \wedge \nu x \in \kappa)$
- $env^{-1}(\ell) = \emptyset.\exists \kappa_i^0 \in \kappa^0 \Rightarrow (\exists \ell^p \in \mathrm{dom}(w).\nu x^p \in \mathrm{dom}(\Gamma) \Rightarrow \nu x \in \kappa_i^0 \wedge \nu x \in \kappa)$

If a value v is a location, then we check that both the set of occurrences agrees with the dependency pair, presented in Definition 15, and check if the alias information agrees with the semantics, Definition 16. If the value v is not a location, then its type can either be an abstraction type or a base type. For the base type, we check that the agreement between the set of occurrences and the dependency pair agrees. If the type is an abstraction, then we check that T_2 agrees with the binding model. We are only concerned about the return type T_2 for abstractions, since if the argument parameter is used in the body of the abstraction, then the dependencies would already be part of the return type.

Definition 17 (Type agreement)**.** We say that

$$(env,v,(w,\sqsubset_w),(L,V)) \models (\Gamma,T)$$

iff

- $v \neq \ell$ and $T = T_1 \to T_2$:
  - $(env,v,(w,\sqsubset_w),(L,V)) \models (\Gamma,T_2)$
- $v \neq \ell$ and $T = (\delta,\kappa)$:
  - $(env,(L,V)) \models \delta$
- $v = \ell$ then $T = (\delta,\kappa)$ where:
  - $(env,(L,V)) \models \delta$
  - $(env,(w,\sqsubset_w),v) \models (\Gamma,\kappa)$

Definition 18 (Environment agreement)**.** We say that $(env,sto,(w,\sqsubset_w)) \models (\Gamma,\Pi)$ if

1. $\forall x \in \text{dom}(env).(\exists x^p \in \text{dom}(w)) \wedge (x^p \in \text{dom}(w) \Rightarrow \exists x^p \in \text{dom}(\Gamma))$
2. $\forall x^p \in \text{dom}(w).x^p \in \text{dom}(\Gamma) \Rightarrow env(x) = v \wedge w(x^p) = (L,V) \wedge \Gamma(x^p) = T.$
   $(env, v, (w, \sqsubset_w), (L,V)) \models (\Gamma, T)$
3. $\forall \ell \in \text{dom}(sto).(\exists \ell^p \in \text{dom}(w)) \wedge (\exists \nu x.\forall p \in \{p' \mid \ell^{p'} \in \text{dom}(w)\} \Rightarrow$
   $\nu x^p \in \text{dom}(\Gamma))$
4. $\forall \ell^p \in \text{dom}(w).\exists \nu x^p \in \text{dom}(\Gamma) \Rightarrow w(\ell^p) = (L,V) \wedge \Gamma(\nu x^p) =$
   $T.(env, \ell, (w, \sqsubset_w), (L,V)) \models T$
5. if $p_1 \sqsubset_w p_2$ then $p_1 \sqsubset_\Pi p_2$
6. $\forall \ell^p \in \text{dom}(w).\exists \nu x^p \in \text{dom}(\Gamma) \Rightarrow \exists p' \in \mathbf{P}.\mathsf{IP}_{\sqsubset_w}(\ell, w) \in \mathsf{IP}_{\Upsilon_{p'}}(\nu x, \Gamma)$

- The agreement for local information only relates the information currently known by *env*, and that the information known by w and Γ agrees, in respect to Definition 17. This is ensured by (1) and (2).
- We similarly handle agreement for the global information known, which is ensured by (3) and (4). Since Γ contains the global information for references, we require that there exists a corresponding internal variable to the currently known locations, by comparing them by program points. We also ensure that the dependency information for a location occurrence agrees with the type of a corresponding internal variable occurrence as given by Definition 17.
- We also need to ensure that Π is a good approximation of the order $\sqsubset_w$ and the greatest binding function for p-chains ensures that we always get the necessary reference occurrences. (5) ensures that the ordering information $\sqsubset_w$ agrees with that of Π.
- We finally need to ensure that for every location known, there exists a corresponding internal variable where, getting the greatest binding of this occurrence, ℓ^p, there exists a program point p', such that looking up all greatest bindings for the p'-chain, there exists an internal variable occurrence that corresponds to ℓ^p. This is captured by (6).

Lemma 1 (History)**.** *Suppose e^p is an occurrence, that*

$$env \vdash \left\langle e^p, sto, (w, \sqsubset_w), p' \right\rangle \to \left\langle v, sto', (w', \sqsubset'_w), (L,V), p'' \right\rangle$$

and $x^{p_1} \in dom(w') \backslash dom(w)$. Then $x \notin fv(e^p)$

Lemma 2 (Strengthening)**.** *If $\Gamma[x^{p'} : T'], \Pi \vdash e^p : T$ and $x \notin fv(e^p)$, then $\Gamma, \Pi \vdash e^p : T$*

4.3 The soundness theorem

We can now present the soundness theorem for our type system.

Theorem 1 (Soundness)**.** *Suppose $e^{p'}$ is an occurrence where*

- $env \vdash \left\langle e^{p'}, sto, (w, \sqsubset_w), p \right\rangle \to \langle v, sto', (w', \sqsubset'_w), (L,V), p'' \rangle,$
- $\Gamma, \Pi \vdash e^{p'} : T$
- $\Gamma, \Pi \vdash env$
- $(env, sto, (w, \sqsubset_w)) \models (\Gamma, \Pi)$

Then we have that

- $\Gamma, \Pi \vdash v : T$
- $(env, sto', (w', \sqsubset'_w)) \models (\Gamma, \Pi)$
- $(env, (w', \sqsubset'_w), v, (L, V)) \models (\Gamma, T)$

Proof. (Outline) The proof proceeds by induction on the height of the derivation tree for

$$env \vdash \left\langle e^{p'}, sto, \psi, p \right\rangle \rightarrow \langle v, sto', \psi', (L, V), p'' \rangle$$

We will only show the proof of four rules here, for (VAR), (CASE), (REF), and (REF-WRITE).

(VAR) Here $e^{p'} = x^{p'}$, where

(VAR)

$$\frac{}{env \vdash \left\langle x^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \rightarrow \left\langle v, sto, (w, \sqsubseteq_w), (L, V \cup \{x^{p'}\}), p' \right\rangle}$$

Where $env(x) = v$, $x^{p''} = uf_{\sqsubseteq_w}(x, w)$, and $w(x^{p''}) = (L, V)$

And from our assumptions, we have:

- $\Gamma, \Pi \vdash x^{p'} : T$
- $\Gamma, \Pi \vdash env$
- $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$

To type the occurrence $x^{p'}$ we use the rule (T-VAR):

(T-VAR)

$$\frac{}{\Gamma, \Pi \vdash x^p : T \sqcup (\{x^p\}, \emptyset)}$$

Where $x^{p''} = uf_{\sqsubseteq_\Pi}(x, \Gamma)$, $\Gamma(x^{p''}) = T$.

We need to show that **1)** $\Gamma, \Pi \vdash c : T$, **2)** $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$, and **3)** $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$.

1) Since, from our assumption, we know that $\Gamma, \Pi \vdash env$, we can then conclude that $\Gamma, \Pi \vdash v : T$

2) Since there are no updates to *sto* and $(w, \sqsubseteq_w)$, we then know that $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$ holds after an evaluation.

3) Since there are no updates to sto' and $(w', \sqsubseteq'_w)$, since $(L, V) = w(x^{p''})$ and since $T = \Gamma(x^{p''})$, we then know that $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$. Due to Definition 17 we can conclude that:

$$(env, v, (w', \sqsubseteq'_w), (L, V \cup \{x^{p''}\})) \models (\Gamma, T \sqcup \{x^{p''}\})$$

(CASE) Here $e^{p'} = \left[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}\right]^{p'}$, where

(CASE)

$$\frac{\begin{array}{c} env \vdash \left\langle e^{p''}, sto, (w, \sqsubseteq_w), p \right\rangle \to \langle v_e, sto'', (w'', \sqsubseteq''_w), (L'', V''), p'' \rangle \\ env[env'] \vdash \left\langle e_j^{p_j}, sto'', (w''', \sqsubseteq''_w), p'' \right\rangle \to \langle v, sto', (w', \sqsubseteq'_w), (L', V'), p_i \rangle \end{array}}{env \vdash \left\langle \left[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}\right]^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \to \langle v, sto', (w', \sqsubseteq'_w), (L, V), p' \rangle}$$

Where $match(v_e, s_i) = \perp$ for all $1 \leq u < j \leq |\tilde{\pi}|$, $match(v_e, s_j) = env'$, and $w''' = w''[x \mapsto (L'', V'')]$ if $env' = [x \mapsto v_e]$ else $w''' = w''$

And from our assumptions, we have that:

- $\Gamma, \Pi \vdash \left[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}\right]^{p'} : T$,
- $\Gamma, \Pi \vdash env$
- $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$,

To type $[\text{case } e^{p''}\ \tilde{\pi}\ \tilde{o}]^{p'}$ we need to use the (T-CASE) rule, where we have:

(T-CASE)

$$\frac{\begin{array}{c} \Gamma, \Pi \vdash e^p : (\delta, \kappa) \\ \Gamma', \Pi \vdash e_i^{p_i} : T_i \quad (1 \leq i \leq |\tilde{\pi}|) \end{array}}{\Gamma, \Pi \vdash [\text{case } e^p\ \tilde{\pi}\ \tilde{o}]^{p'} : T}$$

Where $T = T' \sqcup (\delta, \kappa)$, $T' = \bigcup_{i=1}^{|\tilde{\pi}|} T_i$, $e_i^{p_i} \in \tilde{o}$ and $s_i \in \tilde{\pi}$, and $\Gamma' = \Gamma[x^p : (\delta, \kappa)]$ if $s_i = x$.
We must show that **1)** $\Gamma, \Pi \vdash v : T$, **2)** $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$, and
3) $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$.
To conclude, we first need to show for the premises, where due to our assumption and from the first premise, we can use the induction hypothesis to get:

- $\Gamma, \Pi \vdash v_e : (\delta, \kappa)$,
- $(env, sto'', (w'', \sqsubseteq''_w)) \models (\Gamma, \Pi)$,
- $(env, v, (w'', \sqsubseteq''_w), (L, V)) \models (\Gamma, (\delta, \kappa))$

Since in the rule (T-CASE) we take the union of all patterns, we can then from the second premise:

- $\Gamma, \Pi \vdash v : T_j$,
- $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$,
- $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T_j)$

If we have **a)** $\Gamma', \Pi \vdash env[env']$ and **b)** $(env[env'], sto'', (w''', \sqsubseteq''_w)) \models (\Gamma', \Pi)$, we can then conclude the second premise by our induction hypothesis.

a) We know that either we have $\Gamma' = \Gamma[x \mapsto (\delta, \kappa)]$ and $env[x \mapsto v_e]$ if $s_j = x$, or $\Gamma' = \Gamma$ and env if $s_j \neq x$.
  - if $s_j \neq x$: Then we have $\Gamma, \Pi \vdash env$
  - if $s_j = x$: Then we have $\Gamma[x \mapsto (\delta, \kappa)], \Pi \vdash env[x \mapsto v_e]$, which hold due to the first premise.

b) We know that either we have $\Gamma' = \Gamma[x \mapsto (\delta, \kappa)]$ and $env[x \mapsto v_e]$ if $s_j = x$, or $\Gamma' = \Gamma$ and env if $s_j \neq x$.

- if $s_j \neq x$: then we have $(env, sto'', (w'', \sqsubset''_w)) \models (\Gamma, \Pi)$.
- if $s_j = x$: then $(env[x \mapsto v_e], sto'', (w''', \sqsubset''_w)) \models (\Gamma[x \mapsto (\delta, \kappa)], \Pi)$, since we know that $(env, sto'', (w'', \sqsubset''_w)) \models (\Gamma, \Pi)$, we only need to show for x. Since we have $x \in dom(env)$, $x^{p_j} \in dom(w''')$ and $x^{p_j} \in dom(\Gamma')$ and due to the first premise, we know that $(env[x \mapsto v_e], sto'', (w''', \sqsubset''_w)) \models (\Gamma[x \mapsto (\delta, \kappa)], \Pi)$.

Based on **a)** and **b)** we can then conclude:

1) Since $\Gamma', \Pi \vdash v : T_j$, then we also must have $\Gamma', \Pi \vdash v : T$, since T only contains more information than T_j.

2) By the second premise, Lemma 1, and Lemma 2, we can then get

$$(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$$

3) Due to **1)**, **2)**, **a)**, and **b)** we can then conclude that

$$(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$$

(REF-READ) Here $e^{p'} = [!e_1^{p_1}]^{p'}$, where

(REF-READ)

$$\frac{env \vdash \langle e^{p_1}, sto, (w, \sqsubseteq_w), p\rangle \to \langle \ell, sto', (w', \sqsubseteq'_w), (L_1, V_1), p_1\rangle}{env \vdash \left\langle [!e^{p_1}]^{p'}, sto, (w, \sqsubseteq_w), p \right\rangle \to \left\langle v, sto', (w', \sqsubseteq'_w), (L \cup L_1 \cup \{\ell^{p''}\}, V \cup V_1), p' \right\rangle}$$

Where $sto'(\ell) = v$, $\ell^{p''} = uf_{\sqsubseteq'_w}(\ell, w')$, and $w'(\ell^{p''}) = (L, V)$

And from our assumptions, we have that:

- $\Gamma, \Pi \vdash [!e_1^{p_1}]^{p'} : T$,
- $\Gamma; \Pi \vdash env$
- $(env, sto, (w, \sqsubseteq_w)) \models (\Gamma, \Pi)$,

To type $[!e_1^{p_1}]^{p'}$ we need to use the (T-REF-READ) rule, where we have:

(T-REF-READ)

$$\frac{\Gamma, \Pi \vdash e^p : (\delta, \kappa)}{\Gamma, \Pi \vdash [!e^p]^{p'} : T \sqcup (\delta \cup \delta', \emptyset)}$$

Where $\kappa \neq \emptyset$, $\delta' = \{\nu x^{p'} \mid \nu x \in \kappa\}$, $\nu x_1, \cdots, \nu x_n \in \kappa$.
$\{\nu x_1^{p_1}, \cdots, \nu x_1^{p_m}\} = uf_{\Upsilon_{p'}}(\nu x_1, \Gamma), \cdots, \{\nu x_n^{p'_1}, \cdots, \nu x_n^{p'_s}\} = uf_{\Upsilon_{p'}}(\nu x_n, \Gamma)$, and
$T = \Gamma(\nu x_1^{p_1}) \cup \cdots \cup \Gamma(\nu x_1^{p_m}) \cup \cdots \cup \Gamma(\nu x_n^{p'_1}) \cup \cdots \cup \Gamma(\nu x_n^{p'_s})$.
We must show that **(1)** $\Gamma, \Pi \vdash v : T$, **(2)** $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$, and
(3) $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, T)$.

To conclude, we first need to show for the premises, where due to our assumption and from the premise, we can use the induction hypothesis to get:

- $\Gamma, \Pi \vdash \ell : (\delta, \kappa)$,
- $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$,

- $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, (\delta', \kappa'))$

Due to $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$ and $(env, v, (w', \sqsubseteq'_w), (L, V)) \models (\Gamma, (\delta', \kappa'))$, and due to our assumptions, we can conclude that:

(1) $\Gamma, \Pi \vdash v : T$,

(2) $(env, sto', (w', \sqsubseteq'_w)) \models (\Gamma, \Pi)$,

(3) $(env, v, (w', \sqsubseteq'_w), (L \cup \{\ell^{p''}\}, V)) \models (\Gamma, T \sqcup (\delta \cup \delta', \emptyset))$

□

5 An implementation

We have made an implementation in the Rust programming language. It includes a parser, and evaluator, a type checker, and an approximator for an order of programs points. The approximator is based on the given type system, where it is derived from the structure of the type system. The implementation is hosted on Github and can be found at [4].

6 Conclusion

We have introduced a type system for local data-flow analysis for a subset of ReScript that includes functional as well as imperative feature, notably that of references.

The type system provides a safe approximation of the data flow in an expression. This also allows us to reason about security properties. In particular, the notion of non-interference introduced by Goguen and Meseguer [2] and studied in information-flow analysis can be understood in this setting. A program satisfies the non-interference property if the variables classified as *low* cannot be affected by variables classified as *high*. This corresponds to the absence of chains in Π in which low occurrences appear below high occurrences. A topic of further work is to understand the relative expressive power of our system wrt. the systems of Volpano and Smith [11, 10].

On the other hand, the system contains slack. In particular, the type system is monomorphic. This means that a locally declared abstraction cannot be used at multiple places, even though this may be safe, as this would mean it would contain occurrences at multiple program points. Moreover, abstractions cannot be bound to references.

Polymorphism for the base type (δ, κ) would allow abstractions to be used multiple times in an occurrence. Consider as an example

$(\textbf{let}\ \mathtt{x}\ (\lambda\ \mathtt{y}.\mathtt{y}^1)^2\ (\mathtt{x}^3\ (\mathtt{x}^4\ 1^5)^6)^7)^8$

Occurrences such as this would now become typable, since when typing the applications, the type of the argument changes, as the occurrence x^4 is present in the second application.

The way references are defined currently in the type system, they cannot be bound to abstractions. If this should be introduced a couple of questions need to be evaluated. First there should be looked into base type polymorphism, the second would be to look into type polymorphism, i.e., allow a reference to be bound to an arrow type at one point and a base type at another.

A next step is to devise a type inference algorithm for the type system. An inference algorithm must compute an approximated order of program points, a proper κ_0 and the types for abstractions, that is, find all the places where the parameter of an abstraction should be bound. We conjecture that such a type inference algorithm for our system will be able to compute the information found in an interative data flow analysis.

References

[1] ReScript Association (2020): *BuckleScript and Reason Rebranding*. Available at `https://rescript-lang.org/blog/bucklescript-is-rebranding`.

[2] J. A. Goguen & J. Meseguer (1982): *Security Policies and Security Models*. In: *1982 IEEE Symposium on Security and Privacy*, pp. 11–11, doi:10.1109/SP.1982.10014.

[3] Gary A. Kildall (1973): *A unified approach to global program optimization*. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, Association for Computing Machinery, New York, NY, USA, p. 194–206, doi:10.1145/512927.512945.

[4] Nicky Ask Lund (2023): *Implementation of dataflow analysis*. Available at `https://github.com/loevendallund/dataflow`.

[5] Nicky Ask Lund & Hans Hüttel (2024): *A type system for data flow and alias analysis in ReScript*. Technical Report, Aalborg University. Available at `http://arxiv.org/abs/2408.11954`.

[6] Markus Mohnen (2002): *A Graph-Free Approach to Data-Flow Analysis*. In: *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, Springer-Verlag, Berlin, Heidelberg, p. 46–61. Available at `https://doi.org/10.1007/3-540-45937-5_6`.

[7] Greg Morrisett, Amal J. Ahmed & Matthew Fluet (2005): *L^3: A Linear Language with Locations*. In Pawel Urzyczyn, editor: *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, *Lecture Notes in Computer Science* 3461, Springer, pp. 293–307, doi:10.1007/11417170_22.

[8] Barbara G. Ryder & Marvin C. Paull (1988): *Incremental data-flow analysis algorithms*. *ACM Trans. Program. Lang. Syst.* 10(1), p. 1–50, doi:10.1145/42192.42193.

[9] Frederick Smith, David Walker & Greg Morrisett (2000): *Alias Types*. In Gert Smolka, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 366–381, doi:10.1007/3-540-46425-5_24.

[10] D. Volpano & G. Smith (1997): *Eliminating covert flows with minimum typings*. In: *Proceedings 10th Computer Security Foundations Workshop*, pp. 156–168, doi:10.1109/CSFW.1997.596807.

[11] Dennis M. Volpano & Geoffrey Smith (1997): *A Type-Based Approach to Program Security*. In: *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '97, Springer-Verlag, Berlin, Heidelberg, p. 607–621. Available at `https://doi.org/10.1007/BFb0030629`.

Intuitionistic Propositional Logic in Lean

Dafina Trufaș
LOS, Faculty of Mathematics and Computer Science, University of Bucharest
Institute for Logic and Data Science, Bucharest
dafina.trufas@s.unibuc.ro

In this paper we present a formalization of Intuitionistic Propositional Logic in the Lean proof assistant. Our approach focuses on verifying two completeness proofs for the studied logical system, as well as exploring the relation between the two analyzed semantical paradigms - Kripke and algebraic. In addition, we prove a large number of theorems and derived deduction rules.

1 Introduction

We formalize Intuitionistic Propositional Logic (IPL) using the Lean interactive theorem prover [4]. Our main goal is verifying the soundness and the strong completeness of IPL, with respect to both the Kripke and the Heyting algebras semantics. The language we work with has falsity, conjunction, disjunction and implication as primitive connectives and for syntactical inference we use the Hilbert-style proof system introduced by Gödel in [9].

For the formalization we present in this paper, we chose the Lean proof assistant [4]. An evidence of Lean's proving power and versatility is the Mathlib library [1], maintained by the Lean community. This work aligns with the effort of the Mathlib community to encode mathematical knowledge, and particularly logical systems, in Lean. The underlying theory of Lean is based on a version of dependent type theory, known as the calculus of inductive constructions [3]. Thus, type-checking is the mechanism which assists the user in their approach to prove mathematical statements, either by directly constructing proof terms or by using Lean's so-called tactic-mode.

In the following, we describe the main stages of the implementation and motivate our main design choices. Sections 3.1 and 3.2 describe the formalization of the language and proof-system of IPL. The Kripke completeness proof is based on the so-called canonical model, whose construction relies on the notion of disjunctive theory. Some results about consistent and complete pairs, presented in Section 3.3, are also essential in the flow of this first completeness theorem. In the upcoming Section 3.4, we introduce the Kripke semantics, then in Section 3.5 we present the main steps of the completeness formalized proof with respect to it. Similarly, Section 3.6 proceeds by defining the necessary Heyting algebras notions, establishes the algebraic semantics and concludes by proving the second completeness theorem and establishing the equivalence between the validity notions. Our presentation is inspired by the textbooks of Mints [11], Fitting [5] and Troelstra [2], and the lecture notes of Kuznetsov[10] and Georgescu [7, 6]. All the detailed proofs can be found in my Bachelor's thesis, which is available online at [13].

To the best of our knowledge, the only proof of completeness for IPL formally-verified in Lean is due to Guo, Chen and Bentzen [8]. However, the novelty of our approach consists in:

(i) using a different Hilbert-style proof system;

(ii) proving a large collection of theorems and derived deduction rules;

(iii) formalizing the algebraic semantics of IPL and proving a second completeness theorem, with respect to it;

(iv) implementing a semantic proof of the equivalence between algebraic and Kripke validity;

(v) the manner we dealt with the countability of the set of formulas, which we consider simpler than the method in [8].

2 On the formalization

The Lean code is structured in 8 files, which we briefly describe in the following. First, we have the *Formula.lean* file, which contains the definition of the language (Section 3.1), as well as the proof of the countability of the *Formula* type (Section 3.3). Then, the *Syntax.lean* file proceeds by formalizing the definition of *Proof* (Section 3.2). It includes a large collection of theorems and derived deduction rules, as well as the deduction theorem and some utilitary lemmas. The *Semantics.lean* file contains the definition of the Kripke model, and the semantical definitions we detail in Section 3.4. In the *Soundness.lean* file, the interested reader can find the formalization of the soundness theorem (whose statement we mention in Section 3.5.1), along with an auxiliary lemma used in its proof. Then, *CompletenessListUtils.lean* groups together some utilitary lemmas about *Finset*s of formulas, which are useful when proving some completeness-related theorems. The Kripke completeness theorem, presented in Section 3.5.2, preceded by the definitions and results from Section 3.3, are formalized in the *Completeness.lean* file. Finally, the Heyting algebras notions and necessary results are formalized in the *HeytingAlgebraUtils.lean* file, while the algebraic semantics, culminating with its associated completeness theorem and the equivalence between the validity notions can be found in *HeytingAlgebraSemantics.lean*.

Fragments of Lean proofs will be included in the presentation only if we consider they contain worth-mentioning technical aspects, or, in some cases, in order to sketch the key proof-steps. The full source code is almost 3300 lines long and is available online in [14].

3 Intuitionistic Propositional Logic

In this section, we proceed to describe the main aspects of our formalization. For full theoretical details of the results and proofs, the interested reader may refer to [13].

3.1 Language

We first formalize the countable set of propositional variables, as a wrapper over the *Nat* type. Structures are used to define non-recursive inductive data types, containing only one constructor. And this is also the case here: we can identify any propositional variable with a natural number, so it is convenient to define the *Var* type as a structure with a single field, specifying the index of the variable:

```
structure Var where
  val : Nat
```

We work with a language containing falsity ($\bot$), conjunction ($\land$), disjunction ($\lor$) and implication ($\to$) as primitive logical connectives. Thus, it is natural to define formulas by means of an inductive type, in which the first non-recursive constructor uses the above defined structure type and simply encapsulates it in a *Formula* term, the second is meant to construct falsity, while the following recursive constructors correspond each to one of the primitive connectives:

```
inductive Formula where
| var : Var → Formula
| bottom : Formula
| and : Formula → Formula → Formula
| or : Formula → Formula → Formula
| implication : Formula → Formula → Formula
```

For readability reasons, we introduce the standard Unicode symbol for falsity and define infix notations for the binary connectives, which are much more convenient to use than the S-expressions in which Lean displays the constructors by default. Additionally, we define the derived connectives for equivalence, negation and truth, along with their standard notations:

```
notation "⊥" => bottom
infixl:60 " ∧∧ " => and
infixl:60 " ∨∨ " => or
infixr:50 (priority := high) " ⇒ " => implication

def equivalence (φ ψ : Formula) := (φ ⇒ ψ) ∧∧ (ψ ⇒ φ)
infix:40 " ⇔ " => equivalence

def negation (φ : Formula) : Formula := φ ⇒ ⊥
prefix:70 " ∼ " => negation

def top : Formula := ∼⊥
notation " ⊤ " => top
```

3.2 Proof system

In this formalization, we adhere to the Hilbert-style proof system for IPL introduced by Gödel in [9]. We define this using again an inductive type, with constructors for each axiom and deduction rule:

```
inductive Proof (Γ : Set Formula) : Formula → Type where
| premise {φ} : φ ∈ Γ → Proof Γ φ
| contractionDisj {φ} : Proof Γ (φ ∨∨ φ ⇒ φ)
| contractionConj {φ} : Proof Γ (φ ⇒ φ ∧∧ φ)
| weakeningDisj {φ ψ} : Proof Γ (φ ⇒ φ ∨∨ ψ)
| weakeningConj {φ ψ} : Proof Γ (φ ∧∧ ψ ⇒ φ)
| permutationDisj {φ ψ} : Proof Γ (φ ∨∨ ψ ⇒ ψ ∨∨ φ)
| permutationConj {φ ψ} : Proof Γ (φ ∧∧ ψ ⇒ ψ ∧∧ φ)
| exfalso {φ} : Proof Γ (⊥ ⇒ φ)
| modusPonens {φ ψ} : Proof Γ φ → Proof Γ (φ ⇒ ψ) → Proof Γ ψ
| syllogism {φ ψ χ} : Proof Γ(φ ⇒ ψ) → Proof Γ(ψ ⇒ χ) → Proof Γ(φ ⇒ χ)
| exportation {φ ψ χ} : Proof Γ (φ ∧∧ ψ ⇒ χ) → Proof Γ (φ ⇒ ψ ⇒ χ)
| importation {φ ψ χ} : Proof Γ (φ ⇒ ψ ⇒ χ) → Proof Γ (φ ∧∧ ψ ⇒ χ)
| expansion {φ ψ χ} : Proof Γ (φ ⇒ ψ) → Proof Γ (χ ∨∨ φ ⇒ χ ∨∨ ψ)
```

The notion of Γ-theorem is defined as usual and we denote this by $\Gamma \vdash \varphi$. In Lean, we introduce this notation, as follows:

```
infix:25 " ⊢ " => Proof
```

The above definition of *Proof* generates an elimination rule for this type, which provides us with the formalized mechanisms of the recursion and induction principles on proof terms.
Below we provide an example of how a pen-and-paper formal proof of a derived deduction rule can be

transposed into a mechanized Lean proof:

$$\begin{array}{lll}
(1) & \Gamma \vdash \varphi \wedge \psi \to \varphi & \text{(WEAKENING)} \\
(2) & \Gamma \vdash \varphi \to \varphi \vee \gamma & \text{(WEAKENING)} \\
(3) & \Gamma \vdash \varphi \wedge \psi \to \varphi \vee \gamma & \text{(SYLLOGISM): (1), (2)}
\end{array}$$

```
def disjOfAndElimLeft : Γ ⊢ φ ∧∧ ψ ⇒ φ ∨∨ γ :=
  syllogism weakeningConj weakeningDisj
```

Note that, in the reverse-Hilbert formalized proof, we don't need to pass them explicitly, when constructing the proof term, as the arguments of the constructors in the *Proof* type are implicit, so the Lean kernel will synthesize them from the context.

3.3 Disjunctive theories, consistent and complete pairs

These notions of disjunctive theories, consistent and complete pairs, and some results regarding them are essential in the Kripke completeness proof for IPL, as we will see in Section 3.5.2. Let us recall the definitions of these notions, which can be consulted in [10]. A set of formulas is said to be a disjunctive theory if it is deductively closed ($\Gamma \vdash \varphi$ implies $\varphi \in \Gamma$), consistent ($\Gamma \nvdash \bot$) and disjunctive ($\Gamma \vdash \varphi \vee \psi$ implies $\Gamma \vdash \varphi$ or $\Gamma \vdash \psi$). Then, a pair of sets of formulas (Γ, Δ) is called consistent if there are no $G_1, \ldots, G_n \in \Gamma$ and $D_1, \ldots, D_m \in \Delta$, such that $\vdash G_1 \wedge \ldots \wedge G_n \to D_1 \vee \ldots \vee D_m$. Finally, we say that a consistent pair is complete, if it is a partition of the set of formulas.

```
def dedClosed {Γ : Set Formula} := ∀ (φ : Formula), Γ ⊢ φ → φ ∈ Γ

def consistent {Γ : Set Formula} := Γ ⊢ ⊥ → False

def disjunctive {Γ : Set Formula} :=
    ∀ (φ ψ : Formula), Γ ⊢ φ ∨∨ ψ → Sum (Γ ⊢ φ) (Γ ⊢ ψ)

def disjunctiveTheory {Γ : Set Formula} :=
    @dedClosed Γ /\ @consistent Γ /\ Nonempty (@disjunctive Γ)

def consistentPair {Γ Δ : Set Formula} :=
    ∀ (Φ Ω : Finset Formula), Φ.toSet ⊆ Γ → Ω.toSet ⊆ Δ →
    (∅ ⊢ Φ.toList.foldr Formula.and (∼⊥) ⇒ Ω.toList.foldr Formula.or ⊥ →
    False)

def completePair {Γ Δ : Set Formula} :=
    @consistentPair Γ Δ /\ ∀ (φ : Formula),(φ ∈ Γ /\ φ ∉ Δ) ∨ (φ ∈ Δ /\
    φ ∉ Γ)
```

Below we give the formalized statement of the lemma claiming that given a consistent pair, any formula can be added to one of the sets in the pair, preserving the consistency:

```
lemma add_preserves_cons :
    @consistentPair Γ Δ → ∀ (φ : Formula), @consistentPair ({φ} ∪ Γ) Δ ∨
                                            @consistentPair Γ ({φ} ∪ Δ)
```

The proof of the above lemma follows by reductio ad absurdum and it requires a syntactical derivation, but it doesn't give rise to any technical difficulties, so we do not present it here.

Then, to prove the essential *consistent_incl_complete* lemma, stating that any consistent pair can be component-wise included in a complete one, we define an indexed family of formula-set pairs, thus:

```
def family (nf : Nat → Formula) (n : Nat) : Set Formula × Set Formula :=
    match n with
    | .zero => @add_formula_to_pair Γ Δ (nf 0)
    | .succ n => @add_formula_to_pair (family nf n).fst (family nf n).snd
                 (nf (n + 1))
```

To have access to an enumeration of formulas, we pass as the first argument a function which assigns, to any natural number, a formula. Then, we inductively build the family, by adding the formulas to one of the sets in the pair, whilst preserving the consistency. Without loss of generality, we define the function to add the formula to the first set in the pair, if possible:

```
def add_formula_to_pair (φ : Formula) : Set Formula × Set Formula :=
    if @consistentPair ({φ} ∪ Γ) Δ then (({φ} ∪ Γ), Δ)
    else (Γ, {φ} ∪ Δ)
```

By the *add_preserves_cons* lemma previously presented, it follows easily that applying the above defined *add_formula_to_pair* function repeatedly, starting from a consistent pair, we preserve the consistency of the obtained pairs.

The enumeration of formulas is not required to be bijective, a surjection from *Nat* to *Formula* is sufficient in this case, as we don't have any restriction for adding the formulas only once. Classically, the existence of an injective function from a type α to a type β gives evidence that there is a surjection from β to α. Hence, we define an injective function from *Formula* to *Nat*.
To construct the injection, we use Cantor's pairing function, which we multiply by two, for ease of formalization. For a theretical presentation of Cantor's encoding, refer to Section 1.3.9 in [2].

```
def pairing (x y : ℕ) := (x + y) * (x + y + 1) + 2 * x
```

Then, we associate a numerical identifier to any connective symbol and encode formulas into natural numbers by recursively applying the pairing function on the structure of the formula, as follows:

```
def encode_form : Formula → ℕ
| var v => pairing 0 (v.val + 1)
| bottom => 0
| φ ∧∧ ψ => pairing (pairing (encode_form φ) 1) (encode_form ψ)
| φ ∨∨ ψ => pairing (pairing (encode_form φ) 2) (encode_form ψ)
| φ ⇒ ψ => pairing (pairing (encode_form φ) 3) (encode_form ψ)
```

After proving the injectivity of our encoding function, we are able to define an instance of *Countable* for our *Formula* type. The Mathlib definition of the *Countable* type-class is as follows:

```
class Countable (α : Sort u) : Prop where
    exists_injective_nat' : ∃ f : α → ℕ, Injective f
```

So we immediately define the *Countable* instance for the *Formula* type, based on the proof of the encoding's injectivity:

```
instance : Countable Formula := inject_Form.countable
```

Now, having the surjective enumeration at hand, we can get a step closer to the final construction of the complete pair which includes the initial consistent pair component-wise. We prove that any formula φ is contained in one of the sets of the pair with index $fn(\varphi)$, where by fn we denote the injective encoding of formulas into natural numbers:

```
lemma vp_in_ΓiΔi (φ : Formula) (fn : Formula → Nat) (fn_inj : fn.Injective)
  (nf : Nat → Formula) (nf_inv : nf = fn.invFun) :
   φ ∈ (@family Γ Δ nf (fn φ)).fst \/ φ ∈ (@family Γ Δ nf (fn φ)).snd
```

In Mathlib, the inverse of a function is noncomputably defined as follows:

```
noncomputable def invFun {α : Sort u} {β} [Nonempty α] (f : α → β) :
    β → α :=
    fun y ↦ if h : (∃ x, f x = y) then h.choose else Classical.arbitrary α
```

So this is why we can count on this inverse for any function, regardless of its bijectivity. Notice also that the injectivity of fn gives evidence of $invFun$ being the so-called $left-inverse$.
It is also crucial to prove that the family we defined is increasing:

```
lemma increasing_family {nf : Nat → Formula} (i j : Nat) : i <= j →
    (@family Γ Δ nf i).fst ⊆ (@family Γ Δ nf j).fst /\
    (@family Γ Δ nf i).snd ⊆ (@family Γ Δ nf j).snd
```

Next, we define the component-wise union of the indexed pair-family:

```
def consistent_family_union (_ : @consistentPair Γ Δ) (nf : Nat → Formula) :
    Set Formula × Set Formula :=
    ({φ | ∃ i : Nat, φ ∈ (@family Γ Δ nf i).fst},
     {φ | ∃ i : Nat, φ ∈ (@family Γ Δ nf i).snd})
```

This is finally the witness we make use of when proving the existence of a complete pair, component-wise including our initial consistent one. Of course, before using the family union this way, we have to give evidence that it is indeed a partition of the set of formulas. The increasing property is crucial in achieving this last-mentioned goal.
Finally, we present the formalized statement of the $consistent\_incl\_complete$ lemma:

```
lemma consistent_incl_complete :
  @consistentPair Γ Δ → (∃ (Γ' Δ' : Set Formula), Γ ⊆ Γ' ∧ Δ ⊆ Δ' ∧
  @completePair Γ' Δ')
```

This will be useful when proving the completeness of IPL with respect to the Kripke semantics, which will be subsequently presented.

3.4 Kripke semantics

In the sequel, we define the Kripke semantics. The first definition we need is, of course, that of a Kripke model. We first state this informally, then provide its corresponding formalization. An intuitionistic propositional Kripke model is a tuple (W,R,V), where W is a non-empty set, R is a reflexive and transitive binary relation on W and $V : Var \times W \to \{0,1\}$ is a function assigning truth values to variables. V is assumed to be monotone with respect to R, thus $V(p,w) = 1$ and Rww' implies $V(p,w') = 1$.

```
structure KripkeModel (W : Type) where
    R : W → W → Prop
    V : Var → W → Prop
    refl (w : W) : R w w
    trans (w1 w2 w3 : W) : R w1 w2 → R w2 w3 → R w1 w3
    monotonicity (v : Var) (w1 w2 : W) : R w1 w2 → V v w1 → V v w2
```

We formalize the Kripke model as a parameterized structure, where the parameter W represents the space of worlds. Thus, the worlds of a model are in Lean terms of type W. The first field of the structure models the accessibility binary relation R over terms of type W and V is the valuation function, which takes two arguments - a variable and an inhabitant of type W. Then, the last three fields are meant to formalize the properties of the relation R (reflexivity and transitivity) and the monotonicity of the valuation.

The extended valuation function (on formulas) is defined as follows:

```
def val {W : Type} (M : KripkeModel W) (w : W) : Formula → Prop
| Formula.var p => M.V p w
| ⊥ => False
| φ ∧∧ ψ => val M w φ /\ val M w ψ
| φ ∨∨ ψ => val M w φ \/ val M w ψ
| φ ⇒ ψ => ∀ (w' : W), M.R w w' /\ val M w' φ → val M w' ψ
```

We say that a formula φ is true at a world w of a model M, if $V(\varphi,w) = 1$ and we denote this by $M,w \vDash \varphi$. Then, φ is said to be valid in a model $M := (W,R,V)$, if $M,w \vDash \varphi$, for all $w \in W$. And finally, φ is valid, if it is valid in all the Kripke models. We denote this by $\vDash \varphi$.
Below, we present the formalization of these notions:

```
def true_in_world {W : Type} (M : KripkeModel W) (w : W) (φ : Formula): Prop :=
    val M w φ

def valid_in_model {W : Type} (M : KripkeModel W) (φ : Formula) : Prop :=
    ∀ (w : W), val M w Φ

def valid (φ : Formula) : Prop :=
    ∀ (W : Type) (M : KripkeModel W), valid_in_model M Φ
```

We say that M,w forces Γ (and denote it by $M,w \vDash \Gamma$), if $M,w \vDash \varphi$, for all $\varphi \in \Gamma$.

```
def model_sat_set {W : Type}(M : KripkeModel W)(Γ : Set Formula)(w : W):Prop:=
    ∀ (φ : Formula), φ ∈ Γ → val M w φ
```

Another essential notion is that of local semantic consequence. We say that a formula φ is a local semantic consequence of a set Γ, if for all models M, and all worlds w in M, we have that $M,w \vDash \Gamma$ implies $M,w \vDash \varphi$. We denote this by $\Gamma \vDash \varphi$.

```
def sem_conseq (Γ : Set Formula) (φ : Formula) : Prop :=
    ∀ (W : Type) (M : KripkeModel W) (w : W),
    model_sat_set M Γ w → val M w φ
infix:50 " ⊨ " => sem_conseq
```

Then, a set Δ is forced by Γ, if $\Gamma \vDash \varphi$, for all φ in Δ.

```
def set_forces_set (Γ Δ : Set Formula) : Prop :=
    ∀ (φ : Formula), φ ∈ Δ → Γ ⊨ φ
```

3.5 Kripke completeness theorem

3.5.1 Soundness

The soundness theorem claims that any Γ-theorem is a local semantic consequence of Γ ($\Gamma \vdash \varphi$ implies $\Gamma \vDash \varphi$), for any set of formulas Γ and any formula φ.

In Lean, this statement transposes to:

```
theorem soundness (Γ : Set Formula) (φ : Formula) : Γ ⊢ φ → Γ ⊨ φ
```

The proof is straightforward, so we briefly sketch it here. For full detail, the interested reader shall consult the formalization.
We proceed by induction on *Proof*. For all the axiom cases, we apply an auxiliary lemma asserting that any axiom is valid:

```
lemma axioms_valid (φ : Formula) (ax : Axiom φ) : valid φ
```

Worth-mentioning is also the use of the monotonicity property of the valuation function, in the *exportation* case. We prove this result in *Semantics.lean* and mention here only its formalized claim:

```
lemma monotonicity_val (W : Type) (M : KripkeModel W) (w1 w2 : W) (φ : Formula):
    M.R w1 w2 → val M w1 φ → val M w2 φ
```

3.5.2 Completeness

Theorem. (completeness theorem) For any set of formulas Γ and any formula φ:

$$\Gamma \vdash \varphi \text{ iff } \Gamma \vDash \varphi.$$

The left implication is the soundness theorem, which was already proved in Section 3.5.1. For the reverse implication in the completeness theorem, we appeal to nonconstructive reasoning, proceeding by contraposition. More precisely, we assume by reductio ad absurdum that $\Gamma \nvdash \varphi$ and then construct a Kripke model (the so-called canonical model), which satisfies Γ, but does not satisfy φ. Hence, we get that φ is not a local semantic consequence of Γ, which contradicts our assumption. Our approach follows the Henkin-style completeness proof presented in [10].

We first describe the construction of the canonical model. The domain is set to the type of the disjunctive theories. This *setDisjTh* type is defined as a subtype of the *Set Formula* type, as follows:

```
abbrev setDisjTh := {Γ // @disjunctiveTheory Γ}
```

For the *refl*, *trans*, and *monotonicity* fields of the structure, we have to pass proofs of the set inclusion relation satisfying these properties. These proofs are easily completed, using the corresponding Mathlib theorems. Putting this all together, we have:

```
def canonicalModel : KripkeModel (setDisjTh) :=
{
    R := fun (Γ Δ) => Γ.1 ⊆ Δ.1,
    V := fun (v Γ) => Formula.var v ∈ Γ.1,
    refl := fun (Γ) => Set.Subset.rfl
    trans := fun (Γ Δ Φ) => Set.Subset.trans
    monotonicity := fun (v Γ Δ) => by intros; apply Set.mem_of_mem_of_subset
                                      assumption'
}
```

Apart from lemma *consistent_incl_complete* we have already presented in Section 3.3, the Kripke completeness proof requires also the so-called main semantic lemma. This lemma states that the property of the valuation in the definition of the canonical model, holds also for the extended valuation function on formulas. Thus, it claims that $M_0, \Gamma \vDash \varphi$ if and only if $\varphi \in \Gamma$, for any disjunctive theory Γ and formula φ:

```
lemma main_sem_lemma (Γ : setDisjTh) (φ : Formula) :
    val canonicalModel Γ φ ↔ φ ∈ Γ.1
```

It is worth mentioning that the two implications in this lemma cannot be formalized as independent lemmas, because of the *implication* case, where the proof of the left implication depends on the right implication in the induction hypothesis, and vice versa.
Now we have all the necessary ingredients for the completenss contraposition proof informally presented at the beginning of this section. The formalized completeness statement is the following:

```
theorem completeness {φ : Formula} {Γ : Set Formula} :
    Γ ⊨ φ ↔ Nonempty(Γ ⊢ φ)
```

3.6 Algebraic semantics and completeness theorem

Our approach in the current section is based on the exposition in the textbook [12] and the lecture notes [6, 7]. After establishing the Heyting algebras necessary premises, we move on to defining the algebraic models of IPL and the Lindenbaum-Tarski algebra. Finally, we provide a second completeness proof, with respect to the algebraic semantics and prove the equivalence between the Kripke and algebraic validity.

3.6.1 Heyting algebras

First of all, we shall recall the definition of a Heyting algebra. A Heyting algebra (or pseudo-boolean algebra) is a structure $(H, \vee, \wedge, \rightarrow)$ such that H is a bounded lattice and the following residuation property holds: $a \leq b \rightarrow c$ if and only if $a \wedge b \leq c$. Conventionally, we denote a Heyting algebra by H.
We start by formalizing the general definitions on Heyting algebras. Mathlib contains a definition of the *HeytingAlgebra* type class, which encompasses the conditions a type has to satisfy, in order to have the structure of a Heyting algebra. However, we have to formalize and prove the necessary definitions and results about filters.
We consider a type α for which there is an instance of the Mathlib *HeytingAlgebra* class:

```
variable {α : Type u} [HeytingAlgebra α]
```

Then, we formalize the following main definitions, using the above α type-variable, to represent the domain of the Heyting algebra.
A filter is a nonempty set F, satisfying two conditions: (i) for any $x, y \in F$, $x \wedge y \in F$, (ii) for any $x \in F$ and $y \geq x$, we have that $y \in F$.

```
def filter (F : Set α) := (Set.Nonempty F) ∧ (∀ (x y : α), x ∈ F → y ∈ F →
                          x ⊓ y ∈ F) ∧ (∀ (x y : α), x ∈ F → x ≤ y → y∈F)
```

The filter generated by a set X is the intersection of all the filters which include X.

```
abbrev X_filters (X : Set α) := {F // filter F ∧ X ⊆ F}
def X_gen_filter (X : Set α) := {x | ∀ (F : X_filters X), x ∈ F.1}
```

A filter is called proper, if it doesn't contains the first element of the lattice.

```
def proper_filter (F : Set α) := filter F ∧ ⊥ ∉ F
```

Additionally, a proper filter F is said to be prime, if for all $x, y \in H$, if $x \vee y \in F$, then $x \in F$ or $y \in F$.

```
def prime_filter {α : Type} [HeytingAlgebra α] (F : Set α) :=
    proper_filter F ∧ (∀ (x y : α), x ⊔ y ∈ F → x ∈ F ∨ y ∈ F)
```

Next, we present the central Heyting algebras result, which will be used in a subsequent section, when transiting from an algebraic model to the corresponding Kripke one. It asserts that, given a filter F and an element x which is not in F, there exists a prime filter P including the initial filter, such that x is neither an element of P:

```
lemma super_prime_filter (x : α) (F : Set α) (Hfilter : @filter α _ F)
   (Hnotin : x ∉ F) :
    ∃ (P : Set α), @prime_filter α _ P /\ F ⊆ P /\ x ∉ P
```

In the following, we informally sketch the proof of the above lemma and present key-fragments of its formalization. First of all, we show that the set of all the prime filters not containing x has an upper bound:

```
have Hzorn : ∃ F' ∈ X_filters_not_cont_x x, F ⊆ F' ∧
              ∀ (F'' : Set α), F'' ∈ X_filters_not_cont_x x → F' ⊆ F'' →
                F'' = F'
```

This is achieved by applying Zorn's lemma, which is formalized in Mathlib as follows :

```
theorem zorn_subset_nonempty (S : Set (Set α))
    (H : ∀ (c) (_ : c ⊆ S), IsChain (· ⊆ ·) c → c.Nonempty →
         ∃ ub ∈ S, ∀ s ∈ c, s ⊆ ub) (x)
    (hx : x ∈ S) : ∃ m ∈ S, x ⊆ m ∧ ∀ a ∈ S, m ⊆ a → a = m
```

where *isChain* is a *Prop* deciding whether a given set is totally ordered. The upper bound we are looking for is the union of all the chain's elements. In the rest of the proof, our goal is to prove that this upper bound is a prime filter, and we proceed by contraposition, in doing so. We consider two elements y,z such that $y \notin P$ and $z \notin P$. Then, the first step is showing that $P \subset [P \cup \{y\})$ and its analogous $P \subset [P \cup \{z\})$. Using these auxiliary hypotheses and the maximality of P, we prove that $x \in [P \cup \{y\})$ and $x \in [P \cup \{z\})$. Now, having also this hypothesis at hand, the proof concludes by applying a few well-known Heyting algebras properties, as already shown in the theoretical proof.

The following lemma provides a useful characterization of the filter generated by a set X:

```
lemma gen_filter_prop (X : Set α) :
    X_gen_filter X = {a | ∃ (l : List α), l.toFinset.toSet ⊆ X∧inf_list l≤a}
```

We use this form of the generated filter to obtain an auxiliary result which is necessary for the proof of the above *super_prime_filter* lemma:

```
lemma mem_gen_ins_filter (F : Set α) (Hfilter : filter F) :
    y ∈ X_gen_filter (F ∪ {x}) → ∃ (z : α), z ∈ F /\ x ⊓ z ≤ y
```

Applying this last lemma, the residuation property and a few basic properties of Heyting algebras and filters, we obtain another important result, which will be used when constructing the valuation function of the Kripke model associated to an algebraic one:

```
lemma himp_not_mem (F : Set α) (Hfilter : filter F) (Himp_not_mem : x ⇒ y∉F) :
    y ∉ X_gen_filter (F ∪ {x})
```

The *super_prime_filter* lemma has also a couple of corollaries. The first one states that given an element x different from the last element of the algebra, there exists a prime filter P such that $x \notin P$:

```
lemma super_prime_filter_cor1 (x : α) (Hnottop : x ≠ ⊤) :
    ∃ (P : Set α), @prime_filter α _ P /\ x ∉ P
```

To prove this, we trivially show first that $\{\top\}$ is a filter and then, using the *super_prime_filter* lemma, we obtain the necessary witness.
The second corollary follows immediately from the first one. It claims that intersecting all the prime filters, we obtain the set $\{\top\}$:

```
lemma super_prime_filter_cor2 : Set.sInter (@prime_filters α _) = {⊤} :=
```

This is proved by double inclusion and will be of great importance in an upcoming section, when establishing the connection between the two semantical paradigms.

3.6.2 Algebraic models

An algebraic interpretation in H is a function $\overline{h} : Form \to H$ satisfying the following conditions: $\overline{h}(\bot) = 0$ and, for all $\varphi, \psi \in Form$, $\overline{h}(\varphi \wedge \psi) = \overline{h}(\varphi) \wedge \overline{h}(\psi)$, $\overline{h}(\varphi \vee \psi) = \overline{h}(\varphi) \vee \overline{h}(\psi)$ and $\overline{h}(\varphi \to \psi) = \overline{h}(\varphi) \to \overline{h}(\psi)$.
We formalize the notion of algebraic interpretation as follows:

```
def AlgInterpretation (I : Var → α) : Formula → α
| Formula.var p => I p
| Formula.bottom => ⊥
| φ ∧∧ ψ => AlgInterpretation I φ ⊓ AlgInterpretation I ψ
| φ ∨∨ ψ => AlgInterpretation I φ ⊔ AlgInterpretation I ψ
| φ ⇒ ψ => AlgInterpretation I φ ⇒ AlgInterpretation I ψ
```

An algebraic model is a tuple $(H, \overline{h})$.
We've chosen not to explicitly define the notion of algebraic model in Lean, since it would have implied to adjoin the above defined interpretation function to the type. We considered this redundant, since an algebraic model is uniquely determined by the variable-interpretation function.
A formula φ is true in an algebraic model $(H, \overline{h})$, if $\overline{h}(\varphi) = 1$. We denote this by $(H, \overline{h}) \vDash_{alg} \varphi$. We say that φ is algebraically valid in H, if $(H, \overline{h}) \vDash_{alg} \varphi$, for any algebraic model $(H, \overline{h})$. Finally, φ is called algebraically valid, if φ is algebraically valid in any Heyting algebra H. This is denoted by $\vDash_{alg} \varphi$.

```
def true_in_alg_model (I : Var → α) (φ : Formula) : Prop :=
    AlgInterpretation I φ = Top.Top

def valid_in_alg (φ : Formula) : Prop :=
    ∀ (I : Var → α), true_in_alg_model I φ

def alg_valid (φ : Formula) : Prop :=
    ∀ (α : Type) [HeytingAlgebra α], @valid_in_alg α _ φ
```

A set of formulas Γ is true in an algebraic model, if $(H, \overline{h}) \vDash_{alg} \varphi$ for any $\varphi \in \Gamma$. We denote this by $(H, \overline{h}) \vDash_{alg} \Gamma$. We say that Γ is algebraically valid in H, if $(H, \overline{h}) \vDash_{alg} \Gamma$, for any algebraic model $(H, \overline{h})$. A set Γ is algebraically valid, if it is algebraically valid in any Heyting algebra H. This is denoted by $\vDash_{alg} \Gamma$.

```
def set_true_in_alg_model (I : Var → α) (Γ : Set Formula) : Prop :=
    ∀ (φ : Formula), φ ∈ Γ → AlgInterpretation I φ = Top.top

def set_valid_in_alg (Γ : Set Formula) : Prop :=
    ∀ (I : Var → α), set_true_in_alg_model I Γ

def set_alg_valid (Γ : Set Formula) : Prop :=
    ∀ (α : Type) [HeytingAlgebra α], @set_valid_in_alg α _ Γ
```

We say that φ is an algebraic semantic consequence of Γ, if for any algebraic model $(H,\overline{h})$, $(H,\overline{h}) \vDash_{alg} \Gamma$ implies $(H,\overline{h}) \vDash_{alg} \varphi$. We denote this by $\Gamma \vDash_{alg} \varphi$.

```
def alg_sem_conseq (Γ : Set Formula) (φ : Formula) : Prop :=
    ∀ (α : Type)[HeytingAlgebra α](I : Var → α), set_true_in_alg_model I Γ →
    true_in_alg_model I φ
infix:50 " ⊨ₐ " => alg_sem_conseq
```

3.6.3 Lindenbaum-Tarksi algebra

We define the following equivalence relation on formulas, with respect to a set Γ:

$$\varphi \sim_\Gamma \psi \text{ iff } \Gamma \vdash \varphi \leftrightarrow \psi$$

Let $Form/\sim_\Gamma$ be the quotient set. We denote the equivalence class of a formula φ by $\widehat{\varphi}_\Gamma$. The order relation on $Form/\sim_\Gamma$ is defined as follows: $\widehat{\varphi}_\Gamma \leq_\Gamma \widehat{\psi}_\Gamma$ iff $\Gamma \vdash \varphi \to \psi$.
Then, the quotient set $Form/\sim_\Gamma$ is a Heyting algebra (called the Lindenbaum-Tarksi algebra), where: $\widehat{\varphi}_\Gamma \vee \widehat{\psi}_\Gamma = \widehat{\varphi \vee \psi}_\Gamma$, $\widehat{\varphi}_\Gamma \wedge \widehat{\psi}_\Gamma = \widehat{\varphi \wedge \psi}_\Gamma$, $\widehat{\varphi}_\Gamma \to \widehat{\psi}_\Gamma = \widehat{\varphi \to \psi}_\Gamma$, $\widehat{\bot}_\Gamma$ is the first element and $\widehat{\neg\bot}_\Gamma$ is the last element.
First of all, we formalize the equivalence relation on formulas with respect to Γ, along with its standard infix notation:

```
def equiv (φ ψ : Formula) := Nonempty (Γ ⊢ φ ⇔ ψ)
infix:50 "∼" => equiv
```

Next, we define a setoid instance for our *Formula* type, by providing a proof of the above defined relation being indeed an equivalence relation and then we can move to defining the $\leq, \wedge, \vee, \to$ operations on quotients of this setoid. To define quotient conjunction, disjunction and implication, we make use of the built-in $lift_2$ function, which lifts the corresponding binary functions on formulas, to a quotient on both arguments. We give below only the formalization of quotient conjunction. The other quotient operations are defined in a similar manner.

```
def Formula.and_quot (φ ψ : Formula) := Quotient.mk setoid_formula (φ ⋀⋀ ψ)

def and_quot (φ ψ : Quotient setoid_formula) : Quotient setoid_formula :=
    Quotient.lift₂ Formula.and_quot and_quot_preserves_equiv φ ψ
```

Notice the fact that we have to pass as the second argument of $lift_2$ a proof of our binary operation preserving equivalence. The statement of the corresponding lemma is as follows:

```
lemma and_quot_preserves_equiv (φ ψ φ' ψ' : Formula) : φ ∼ φ' → ψ ∼ ψ' →
    (Formula.and_quot φ ψ = Formula.and_quot φ' ψ')
```

Having these operations defined, we can prove that the quotient type associated to the $\sim$ equivalence relation is a Heyting algebra. We do so by defining a Heyting algebra instance for this type:

```
instance lt_heyting : HeytingAlgebra (Quotient (@setoid_formula Γ))
```

We don't provide the full definition of this instance here, but all the proofs we need to complete its fields are rather trivial.
We define the mapping which associates to a formula its corresponding quotient:

```
def h_quot_var (v : Var) : Quotient (@setoid_formula Γ) :=
    Quotient.mk setoid_formula (Formula.var v)

def h_quot (φ : Formula) : Quotient (@setoid_formula Γ) :=
    Quotient.mk setoid_formula φ
```

The *h_quot_var* function will be passed as an argument to *AlgInterpretation*, when proving that *h_quot* satisfies the conditions of an algebraic interpretation. The statement of this lemma is as follows:

```
lemma h_quot_interpretation : ∀ (φ : Formula),  h_quot φ = (@AlgInterpretation
(Quotient (@setoid_formula Γ)) _ h_quot_var φ)
```

Then, we are able to prove the two results about the Lindenbaum-Tarski algebra, which will be crucial in the proof of the algebraic completeness theorem. The first one asserts that a set Γ is true at the algebraic model generated by itself, whilst the second claims that a formula φ is true at the algebraic model induced by Γ, if and only if φ is a Γ-theorem. We mention only their statements below, as the proofs do not contain any technical difficulties:

```
lemma set_true_in_lt :
    @set_true_in_alg_model (Quotient (@setoid_formula Γ)) _ h_quot_var Γ

lemma true_in_lt (φ : Formula) :
    @true_in_alg_model (Quotient (@setoid_formula Γ)) _ h_quot_var φ ↔
    Nonempty (Γ ⊢ φ)
```

3.6.4 Algebraic completeness theorem

Theorem. [algebraic completeness] For any set of formulas Γ and any formula φ,

$$\Gamma \vdash \varphi \text{ iff } \Gamma \vDash_{alg} \varphi.$$

The soundness implication follows immediately, by a straightforward induction. We mention only its formalized statement here:

```
theorem soundness_alg (φ : Formula) : Nonempty (Γ ⊢ φ) → alg_sem_conseq Γ φ
```

Moving now to the reverse implication, the proof is based on the two results mentioned at the end of Section 3.6.3. Below, we present the full formalization of the algebraic completeness theorem:

```
theorem completeness_alg (φ : Formula) :
  alg_sem_conseq Γ φ ↔ Nonempty (Γ ⊢ φ) :=
    by
      apply Iff.intro
      · intro Halg
        rw [←true_in_lt]
        exact Halg (Quotient (@setoid_formula Γ)) h_quot_var set_true_in_lt
      · exact soundness_alg φ
```

3.6.5 Kripke models and algebraic models

The central result in this last section is the equivalence between the two validity notions:

$$\vDash \varphi \text{ iff } \vDash_{alg} \varphi$$

We follow the approach in [5] and hence give a pure semantical proof of the above mentioned result, wihtout using the completeness theorems of the two semantics. We start by establishing a connection from Kripke models to algebraic models. In doing so, we have to define first the notions of closed set, and the Heyting algebra structure which can be built on top of the set of all the closed sets.
Thus, the following *Prop* decides whether a domain set of a Kripke model is closed:

```
def closed {W : Type} (M : KripkeModel W) (A : Set W) : Prop :=
    ∀ (w w' : W), w ∈ A → M.R w w' → w' ∈ A
```

We formalize the set of all closed subsets as a subtype of the *Set W* type, as follows:

```
def all_closed {W : Type} (M : KripkeModel W) := {A // @closed W M A}
```

For the implication operation on closed subsets, we first define the set of all closed sets contained in $W \setminus A \cup B$, where by A, B we denote the two implication operands. Then, the union of the elements in this set is the greatest closed set satisfying our condition:

```
def all_closed_subset {W : Type} (M : KripkeModel W) (A B : all_closed M) :=
    {X | @closed W M X /\ X ⊆ ((@Set.univ W) \ A.1) ∪ B.1}


def himp_closed {W : Type} {M : KripkeModel W} (A B : all_closed M) :=
    Set.sUnion (@all_closed_subset W M A B)
```

We define the corresponding Heyting algebra instance, as follows:

```
instance {W : Type} (M : KripkeModel W) : HeytingAlgebra (all_closed M) :=
  { sup := λ X Y => {val := X.1 ∪ Y.1, property := union_preserves_closed X Y}
    le := λ X Y => X.1 ⊆ Y.1
    le_refl := λ _ => Set.Subset.rfl
    le_trans := λ _ _ _ => Set.Subset.trans
    le_antisymm := λ _ _ => by rw [Subtype.ext_iff]; apply Set.Subset.antisymm
    le_sup_left := λ X Y => Set.subset_union_left X.1 Y.1
    le_sup_right := λ X Y => Set.subset_union_right X.1 Y.1
    sup_le := λ _ _ _ => Set.union_subset
    inf := λ X Y => {val := X.1 ∩ Y.1, property := inter_preserves_closed X Y}
    inf_le_left := λ X Y => Set.inter_subset_left X.1 Y.1
    inf_le_right := λ X Y => Set.inter_subset_right X.1 Y.1
    le_inf := λ _ _ _ => Set.subset_inter
    top := {val := @Set.univ W, property := univ_closed}
    le_top := λ X => Set.subset_univ X.1
    himp := λ X Y => {val := himp_closed X Y, property := himp_is_closed X Y}
    le_himp_iff := λ X Y Z => himp_closed_prop Y Z X
    bot := {val := ∅, property := empty_closed}
    bot_le := λ X => Set.empty_subset X.1
    compl := λ X => {val := himp_closed X {val := ∅, property := empty_closed},
                     property := himp_is_closed X {val := ∅,
                                                   property := empty_closed}}
    himp_bot := by simp }
```

The next step is proving that the following function is an algebraic interpretation:

```
def h {W : Type} {M : KripkeModel W} (φ : Formula) : all_closed M :=
    {val := {w | val M w φ}, property := by intro w w' Hwin Hr
                                            apply monotonicity_val
                                            assumption'}
```

Except for the implication case, the proof is trivial. We present here the main steps of this last interesting case. The proof is by double inclusion, but before succeeding in doing so, we need to prove an additional statement, which holds only for closed subsets:

```
have Haux : ∀ (A : all_closed M),
    A.1 ⊆ (@h W M (ψ ⇒ χ)).1 ↔ A.1 ∩ (@h W M ψ).1 ⊆ (@h W M χ).1
```

By this point, we can formalize the first central result of the section, which provides a method of constructing an algebraic model corresponding to a given Kripke model:

```
lemma kripke_alg {W : Type} {M : KripkeModel W} (φ : Formula) :
  valid_in_model M φ ↔ @true_in_alg_model (all_closed M) _ h_var φ
```

In the sequel, we aim to formalize also the reverse direction, namely the switch from an algebraic model to a corresponding Kripke one. We first define the Kripke frame based on the set of all prime filters. The accessibility relation is given by inclusion and a variable is said to be true at a world of a prime filter *F*, if it is an element of *F*:

```
def prime_filters_frame (I : Var → α) :
    KripkeModel (@prime_filters α _) :=
    {
        R := λ (F1 F2) => F1.1 ⊆ F2.1,
        V := λ (v F) => I v ∈ F.1,
        refl := λ (F) => Set.Subset.rfl,
        trans := λ (F1 F2 Φ) => Set.Subset.trans,
        monotonicity := λ (v F1 F2) => by intros
                                        apply Set.mem_of_mem_of_subset
                                        assumption'
    }
```

and prove that the function given by:

```
def Vh (φ : Formula) (F : @prime_filters α _) (I : Var → α) : Prop :=
    AlgInterpretation I φ ∈ F.1
```

is a valuation function for this frame.
Now, we can state and prove the second relation between algebraic and Kripke models:

```
lemma alg_kripke (I : Var → α) (φ : Formula) :
  true_in_alg_model I φ ↔ valid_in_model (prime_filters_frame I) φ
```

Finally, having this auxiliary results at hand, we can immediately prove the equivalence between Kripke and algebraic validity:

```
theorem alg_kripke_valid_equiv (φ : Formula) :
  alg_valid φ ↔ valid φ :=
    by
      apply Iff.intro
```

```
· intro Halg _ _
  rw [kripke_alg]; apply Halg
· intro Hvalid _ _ _
  rw [alg_kripke]; apply Hvalid
```

4 Conclusion and future work

We have used the Lean proof assistant to formally verify the completeness of IPL. After defining the language, we formalized the Hilbert-style proof system and used it to establish a collection of syntactic theorems and derived deduction rules. The next crucial step was formally specifying the two studied semantics: Kripke and algebraic. For the proof of the completeness theorem with respect to the Kripke semantics, we defined the so-called canonical model, and used it in order to complete the proof by contraposition. On the other hand, for the algebraic completeness proof, we made use of the Lindenbaum-Tarski algebra and some of its specific properties.

As future work, we aim to extend the current formalization to express Intuitionistic First-Order Logic and also provide a completeness proof for this more complex system. Furthermore, we intend to implement in Lean formal systems for intuitionistic arithmetical analysis and associated proof interpretations, as the ones presented in [2].

5 Acknowledgements

The author thanks Laurențiu Leuștean and Traian Șerbănuță for providing comments and suggestions that improved the final version of the paper.

References

[1] *A Mathlib Overview*. Available at `https://leanprover-community.github.io/mathlib-overview.html`.

[2] (1973): *Metamathematical Investigation Of Intuitionistic Arithmetic And Analysis*. In A. S. Troelstra, editor: *Lecture Notes in Mathematics*, 344, Springer, Berlin Heidelberg, doi:10.1007/BFb0066739.

[3] T. Coquand & G. Huet (1988): *The Calculus of Constructions*. *Information and Computation* 76(2-3), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[4] L. De Moura, S. Kong, J. Avigad, F. Van Doorn & J. von Raumer (2015): *The Lean theorem prover (system description)*. In A. Felty & A. Middeldorp, editors: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, *Lecture Notes in Computer Science* 9195, Springer, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[5] M. C. Fitting (1968): *Intuitionistic Logic, Model Theory and Forcing*. Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, doi:10.2307/2271564.

[6] G. Georgescu (1995): *Notes on Heyting Algebras (in Romanian)*. Lecture Notes, University of Bucharest.

[7] G. Georgescu (1995): *Notes on Intuitionistic Logic (in Romanian)*. Lecture Notes, University of Bucharest.

[8] H. Guo, D. Chen & B. Bentzen (2023): *Verified completeness in Henkin-style for intuitionistic propositional logic*. In B. Bentzen, B. Liao, D. Liga, R. Markovich, B. Wei, M. Xiong & T. Xu, editors: *Logics for AI and Law*, College Publications, London, pp. 36–48, doi:10.48550/arXiv.2310.01916.

[9] K. Gödel (1958): *Über eine bisher noch nicht benutzte Erweiterung des finiten Standpunktes*. *Dialectica* 12, pp. 280–287, doi:10.1111/j.1746-8361.1958.tb01464.x.

[10] S. Kuznetsov (2017): *Propositional Intuitionistic Logic*. Lecture Notes, University of Pennsylvania.

[11] G. Mints (2000): *A Short Introduction To Intuitionistic Logic*. The University Series in Mathematics, Kluwer, New York, doi:`10.1007/b115304`.

[12] H Rasiowa & R Sikorski (1963): *The mathematics of metamathematics*. Panstwowe Wydawnictwo Naukowe, Warsaw, doi:`10.1112/jlms/s1-41.1.572`.

[13] D. Trufaș (2024): *Intuitionistic Logic in Lean*. Bachelor Thesis, University of Bucharest. Available at `https://github.com/DafinaTrufas/Intuitionistic-Logic-Lean`.

[14] D. Trufaș (2024): *Intuitionistic Logic in Lean*. `https://github.com/DafinaTrufas/Intuitionistic-Logic-Lean`.

Leveraging Slither and Interval Analysis to build a Static Analysis Tool

Ștefan-Claudiu Susan
Alexandru Ioan Cuza University of Iași,
Department of Computer Science
Iași, România
claudiu_susan@yahoo.com

Even though much progress has been made in identifying and mitigating smart contract vulnerabilities, we often hear about coding or design issues leading to great financial losses. This paper presents our progress toward finding defects that are sometimes not detected or completely detected by state-of-the-art analysis tools. Although it is still in its incipient phase, we developed a working solution built on top of Slither that uses interval analysis to evaluate the contract state during the execution of each instruction. To improve the accuracy of our results, we extend interval analysis by also considering the constraints imposed by specific instructions. We present the current solution architecture in detail and show how it could be extended to other static analysis techniques, including how it can be integrated with other third-party tools. Our current benchmarks contain examples of smart contracts that highlight the potential of this approach to detect certain code defects.

1 Introduction

In the aftermath of the 2008 financial crisis, the population started losing faith in banks, financial governing authorities, and fiat currencies. This was the perfect context for introducing Bitcoin[19], the first cryptocurrency and the first practical implementation and use case for Blockchain technology. It was in complete opposition with the traditional financial instruments, promising transparency, decentralization, and immutability.

Unlike Bitcoin, which was a purely financial implementation of blockchain, Ethereum [7] also supported deploying programs on its network, making it a Blockchain Software Platform. These programs are known as smart contracts. As the name suggests, smart contracts contain a digital form of an agreement made between two or more parties. In contrast to traditional contracts, due to their automatic enforcement of terms, they do not require any trust between the involved parties. Along with the automation capabilities specific to conventional software products, the immutability, transparency, and decentralization of the blockchain make smart contracts the perfect fit for such a use case. The most popular programming language for implementing such contracts is Solidity [1].

As with any incipient technology, Solidity and smart contracts in general had many faults when first introduced. These defects were quickly exploited by malicious users. These issues mainly came from the language design. Two such example are “The DAO Hack“[18] and “Parity Wallet Multisig Bug“[21] Incidents such as these motivated researchers and other people in the community to start discovering and classifying the types of vulnerabilities that can appear in smart contracts. These defects include the ones common among “classic“ programming languages such as `Division by zero` or `Integer`

[1]Solidity, version 0.8.26: `https://docs.soliditylang.org/en/v0.8.26/`

`Underflow/Overflow`, as well as bugs that are specific to the Solidity language and the Blockchain environment, such as `Reentrancy` and `Transaction State Dependency`. Taxonomies such as the one presented in [22] were created with the purpose of offering a better understanding of what bugs might appear. Community-maintained taxonomies such as SWC Registry[2] and DASP Top 10[3] must also be mentioned as notable efforts in this area.

In addition to classifying the most common issues, a lot of effort was also put into starting to manually audit smart contracts and developing tools capable of automatically detecting bugs. These tools implement a variety of approaches to signal code defects, such as symbolic execution and static program analysis. Such implementations include Slither[12], a tool that implements a variety of detectors, Securify[27], a static analysis tool that models good and bad coding patterns as logic formulas; Solhint[2], a linter for the Solidity language; Mythril[25], a symbolic execution approach that analyzes EVM (Ethereum Virtual Machine) Bytecode. Even though not an analysis tool in itself, we must also mention the static analysis plugin that Remix [4], the most popular IDE for Ethereum development, offers. During a previous study we conducted[5], Slither emerged as the overall best-performing tool. Even though it had the highest overall score, there were still a considerable amount of vulnerabilities that were not detected by it or any other tool included in our study. This made us analyze the way it is implemented in greater detail and research methods to aid it in providing more robust detections.

In this paper, we present our current progress in implementing a new analysis tool that is built on top of Slither and leverages the information provided by it after parsing the contract. Our current approach is based on *interval analysis*, with the possibility of being extended in the future. Using this technique, we can approximate the range of values of each variable during each program point, regardless of whether it is a state variable, parameter, local variable, or built-in variable. Having these approximations, we are then able to detect possible coding or design defects based on program state and constraints. A good example of such issues is an unreachable program statement; if a `require` statement contains a condition that can never be fulfilled, then all the instructions after that will not be executed. This also applies to unreachable if branches or loop bodies. Even though our tool is able to approximate the program state for a Solidity function with a reasonable degree of accuracy, it has certain limitations. One such limitation lies in the fact it is not yet able to interpret every instruction in Solidity. Currently, our tool only supports intraprocedural analysis. Another, more specific to interval analysis, is the limited accuracy of the approximation without user input.

Summary of contributions

1. We provide an in-depth explanation of how Slither can be extended and its main modules.
2. We present the architecture that we implemented and the means used to connect with external modules.
3. We provide a collection of smart contracts that our tool can run against and evaluate our solution.

Paper organisation Section 2 contains a summary of our previous work in this area as well as a short presentation of other state-of-the-art analysis tools. Section 3 a briefing of the Static Program Analysis theories that serve as the building blocks of our work. In Section 4, we describe how Slither can be used in a custom implementation and describe the main modules and data types included. We present our

[2]SWC Registry:`https://swcregistry.io/`

[3]DAPS Top 10 `https://dasp.co/`

[4]Remix IDE: `https://remix-project.org/`

solution architecture in Section 5. We evaluate our solution in Section 6 and present its limitations in Section 7. The paper concludes with Section 8.

2 Related work

We already presented the fundamentals of our approach in our initial progress report [26]. It serves as the building blocks of our current implementation. Even though we are still facing some of the limitations outlined during our previous report, we managed to overcome a few important limitations, such as handling conditional, repetitive, and the most widely used builtin statements. In addition, our current implementation is considerably more robust and easy to extend.

There are a number of remarkable tools, besides Slither, that perform static analysis while implementing different approaches, among them, we must mention:

1. Securify[27] is a smart contract security scanner that heavily relies on First Order Logic concepts and formulas to detect issues. It defines multiple patterns as logical formulas and checks program statements to see if any of them match. There are compliance and violation patterns. If a program section matches a compliance pattern, it is not flagged as a defect. If it matches a violation pattern, it is marked as an error with a high degree of certainty. If none of the previous situations apply, it is marked as a warning.
2. Solhint[2] is an open-source linter for the Solidity language. It can be installed via `npm` while also being available as a plugin for numerous IDEs. Its behaviour relies on detection rules set by the user. These guidelines include *Security Rules*, *Style Guide Rules* and *Best Practices Rules*. It requires a configuration file that declares which detection rules must be employed. A default configuration can be generated using a `npm` command.
3. Remix[1] is a dedicated IDE for developing DApps (Decentralized applications). It provides many useful features that can aid in implementing such applications. Besides the compiler whose version can be selected by the user, it also features an emulated version of the Ethereum network. By default, there are 10 available addresses, each holding 100 Ether, any of these addresses can be used to deploy the contracts. After deployment, any address can be used to call functions located in those contracts. Besides integration with Solhint and Slither, it also features its own static analysis plugin. We were unable to find any details regarding the techniques employed in its implementations. During a previous benchmark that we conducted on a number of analysis tools, we found that the static analysis plugin for Remix yielded notable results compared to most tools.

Although it relies more on symbolic and concrete execution, Hevm[11] also aims to detect reachable and unreachable final states for a smart contract function. This solution determines the final states from symbolic inputs. The final states are statically determined to be unreachable or marked as potentially reachable. For the former, SMT queries are generated and passed to a solver. An approach similar in the use of a solver to check the satisfiability of states using contracts is also featured in our solution. Hevm can also check the equivalence of two different implementations for the same functionality by checking the equivalence of final states.

We must also mention Simbolik[5]. Even though it is a debugging tool that uses symbolic execution, not a static analysis code defect detection tool, it needs to be mentioned due to its capabilities of approximating the program state for any step. Leveraging the capabilities of K Framework[23], it can be used for debugging, symbolic testing, and detecting path conditions.

[5]Simbolik is available at `https://simbolik.runtimeverification.com/`

3 Background

3.1 Dataflow Analysis

Dataflow Analysis[4] is a program analysis paradigm that was developed in the context of optimizations performed by compilers and it remains one of its most prominent use cases [15]. Being a static analysis technique, it aims to simulate the behaviour of a program at runtime without executing it. As the name suggests, it analyzes the flow of information through the control flow graph of a program by collecting data about the possible values and states of variables and expressions at various points of a program.

We should clearly differentiate it from *control-flow-analysis*[3]. It clearly determines the order of instructions that are executed in an algorithm given a certain context. They are different in their purpose, but they rely on each other's findings. To properly determine the flow of data, the order of operations is needed as it influences the concrete values at concrete program locations. On the other hand, to determine the order of operations when runtime decisions are present, an approximation of data state is necessary.

Based on different characteristics, data-flow analysis techniques can be divided into different categories. Some of the classification criteria are the following[20]:

- Depending on the direction of flow: **top-down (forward) analysis** that follows the actual execution flow of the program, or **bottom-up (backward) analysis** that does the opposite,
- Depending on the scope: **intraprocedural analysis** that only considers a single function and **interprocedural analysis** that takes into consideration the interaction between functions,
- Depending on program flow consideration: **flow-sensitive analysis** that considers program flow and **flow-insensitive analysis** that does not take it into consideration,
- Depending on how execution paths intersection is treated: **may analysis** that performs program state union and **must analysis** that performs intersection.

Examples of data-flow analysis techniques include, but are not limited to, *Live Variables*, *Available Expressions*, *Constant Propagation*, etc. described in more detail in [24]. A data-flow analysis instance can be formalized as a system containing the following elements:

Given a program P, let $G(P) = (V, E, cmd)$ be the control flow graph of P where:

- V is the set of vertex from the graph,
- $E \subseteq V \times V$ is the set of edges from the graph,
- cmd is the set of statements from P.

A data-flow system $S = (Lab, Ext, Flw, (D, \sqsubseteq), \varepsilon, \varphi)$

- A set of program labels $Lab = V$ in $G(P)$. For example, the numerical labels attributed to each vertex of the control flow graph,
- A subset of extremal labels, *Ext*, the labels where the program starts or where it ends depending on the flow. The nodes marked with extreme labels either have only outbound or only inbound edges,
- A flow relation between labels, a function that models the transition between the program points. *Flw*: $Flw = E$ for forward analysis and $Flw = E^{-1}$ for backward analysis,
- A complete lattice[14] $(D, \sqsubseteq)$ containing all the possible state values, this will serve as the domain of analysis information values. It must contain $\bot$ (bottom) and $\top$ (top) values. These elements are the greatest-lower and least-upper bounds of D,

- An extremal value $\varepsilon \in D$ for extremal labels,
- A collection of transfer functions $\varphi_\ell : D \to D$, $\ell \in Lab$. These functions reflect the changes in a program's state produced by a statement.

Even though the analysis information is not the same for all analysis types, it can be represented as key-value pairs, where the key is the label of the program point and the value is the state representation relevant to that type of analysis. In other words, the analysis information for the whole program contains a collection of representations for each statement/node, which in turn contains a collection of representations for each variable.

This type of static analysis works by presenting the program state as a system of equations and trying to calculate the least fixed points[17]. To accomplish this, several algorithms have been developed, including the MOP (Meet Over all Paths) [16] and the Worklist [20]. We will focus on the latter since it is also the one that our solution implements.

It works by iterating over the control flow graph until no changes occur in the program state. The worklist is initialized with the extremal labels and checks adjacent nodes according to the flow function. Initially, the algorithm could run for an infinite number of iterations for some programs, especially the ones containing loops. This happens when the complete lattice in the data-flow system does not satisfy ACC (Ascending Chain Condition).

We consider that a complete lattice $(D, \sqsubseteq)$ satisfies ACC if the sequence $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots \subseteq d_n$ eventually stabilizes. The chain stabilises if $d_n = d_{n+1} = \cdots$ for some $n \geq 0$. Knowing this, we can certainly say that the algorithm will complete after $|Lab| * n$ iteration at most. Non satisfying ACC domains do not grant this certainty and the algorithm might run indefinitely.

This issue was addressed by using widening operators[8]. Even though they provided an over-approximation, their use made sure that the algorithm runs over a finite number of iterations. To obtain a more precise solution, narrowing operators have been employed. Using them, a more precise approximation can be obtained.

3.2 Interval Analysis

Interval analysis[13][9] is a technique of static analysis that aims to approximate the value of each program variable using intervals. The classic approach to interval analysis uses numeric intervals to represent the range of values of variables at each point in the program.

It is an instance of abstract interpretation[10][6]. It abstracts precise numerical values into intervals. With a simplified domain and operations, this technique can be used to effectively reason a program's potential behavior without necessitating concrete execution and exact values.

A data-flow system $S = (Lab, Ext, Flw, (D, \sqsubseteq), \varepsilon, \varphi)$ for interval analysis using numeric intervals, where Varis the set of variables from the program and *Int* is the set of numeric intervals, is the following:

- *Lab* : *Lab*=V in G(P),
- *Ext*: *Ext*=$\{V \mid \nexists V_i, V_i \to V \in Edges\}$ (program starting point, forward analysis),
- *Flw*: $Flw = Edges$ (forward analysis),
- $(D, \sqsubseteq)$: D=$\{AI \mid AI : Var \to Int\}$, $AI_1 \sqsubseteq AI_2$ iff $AI_1(x) \subseteq AI_2(x)$ for every $x \in Var$. The analysis information for a program point contains the variables that are accessible from the scope of that statement. It maps each variable to a numeric interval. This domain only contains all possible values for `int` and `bool`. To be able to handle a program with more complex types, this domain must be extended,

- $\varepsilon = [-\infty, \infty]$. This is the most general approximation for variables that hold numerical values. In practice, depending on the data type, this interval can be narrowed down. For example, $\varepsilon = [0, \infty]$ for unsigned data types and $\varepsilon = [0(false), 1(true)]$ for boolean values,
- the transfer function:

$$\varphi_\ell = \begin{cases} AI & \text{if } \ell \text{ does not modify state} \\ AI[x_i = [v, v]] & \text{if } \ell \text{ modifies the variable } x_i \text{ with the value } v \end{cases}$$

To further improve the accuracy of our approximations, we can make use of constraints imposed by certain statements. Such instructions are common among most programming languages, such as the `if` statement through its conditional branches and loops via their invariant condition. We must also mention that Solidity has two built-in instructions that impose constraints: `require` and `assert`.

In the domain defined below, *Val* holds the variables state and *Con* is the set of constraints for that point in the program.

To be able to handle complex types, we must extend the codomain of *Val*, we will note it as *IntExt*. In addition to numeric intervals, we must also add mappings between collection indexes or struct fields and numerical intervals. This means that we need to define *IntExt* recursively.

Considering a variable v that has a data type *Type*, we define $Val(v)$ as follows:
The possible values for *Type* are a subset of the data types included in Solidity [6]..
$Type = int \mid bool \mid Struct[f1 : Type_1, ..., fn : Type_n] \mid Array[Type] \mid Map[Type_1, Type_2]$.

- $Val(\texttt{int}) = Int$,
- $Val(\texttt{bool}) = \{[0,0], [1,1], [01]\}$,
- $Val(\{f_1 : T_1, \ldots, f_n : T_n\}) = \{\{\{f_1 : v_1, \ldots, f_n : v_n\} \mid v_i \in Val(T_i), i = \overline{1,n}\}$ where $f_1, ..., f_n$ are all the fields defined in the structure,
- $Val(Array[T]) = \{(i_1 : v_1, \ldots, i_n : v_n) \mid i_j \in Val(\texttt{int}), v_j \in Val(T), j = \overline{1,n}, n \geq 1\}$ and n is the length of the array. If the array has been declared without a size and has not been initialized, we consider n=0,
- $Val(Map[T_1, T_2]) = \{(k_1 : v_1, \ldots, k_n : v_n) \mid k_j \in Val(T_1), v_j \in Val(T_2), j = \overline{1,n}, n \geq 1\}$ where $k_1, .., k_n$ are the keys that have been assigned a value up to that point of the program.

For reference types that contain fields of various data types, Val is recursively applied to each field according to it's data type. We must also extend $\subseteq$(inclusion), $\cup$(reunion) and $\cap$(intersection) for our new domain. For complex types, these operators will be applied in two steps. First, on the keys, and then on the underlying values.
We will consider K as the set of keys for a complex data structure. For example, $K(s)$ if s has the data type of `Struct` is the set of all fields declared in that struct. For mapping and arrays, the result of K is the set of keys/indexes that have been initialized up to that point. In addition, for arrays of fixed length, the result of K will
The behavior for these extended operations is the following:

- $Val(x_1) \subseteq Val(x_2)$ if $K(x_1) \subseteq K(x_2)$ and $Val(x_1)[k_i] \subseteq Val(x_2)[k_i] \forall k_i \in K(x_1)$
- $Val(x_1) \cap Val(x_2) = \{k_i : Val(x_1)[k_i] \cap Val(x_2)[k_i]\} \forall k_i \in K(x_1) \cap K(x_2)$

[6]Solidity 0.8.26 data types: `https://docs.soliditylang.org/en/v0.8.26/types.html`

- $Val(x_1) \cup Val(x_2) = \{k_i : Val(x_1)[k_i] \cup Val(x_2)[k_i]\} \forall k_i \in K(x_1) \cup K(x_2)$. If k_i is not included in $K(x_1)$ and is included in $K(x_2)$ then only the value $Val(x_2)[k_i]$ will be considered when computing the result and vice-versa.

Considering *BExp* as the set of boolean expressions present in the program, our domain becomes the following:

$$\text{D}=\{AI = (Val, Con) \mid Val : Var \to IntExt, C \subseteq BExp\},$$
$$AI_1 \sqsubseteq AI_2 \text{ iff } Val_1(x) \subseteq Val_2(x) \text{ for every } x \in Var, Con_1 \implies Con_2.$$
$$AI_1 \cup AI_2 = (Val_1 \cup Val_2, Con_1 \cup Con_2)$$

The constraints are defined over symbolic variables. They are propagated only downward when traversing the CFG. Currently, Solidity does not feature a `Goto` instruction or any other jump statement. Due to this, the only possible cycles that can be present in the CFG are the ones created by loops. To mediate cases where a node can be reached from multiple paths, we only impose constraints on the starting point of the loop that come from the initial path of execution, we do not add the constraints imposed during the loop body or the loop invariant. We consider that this approach is the most suitable for our use case since it ensures termination when the program includes loops. Although it might lead to a loss of precision because fewer constraints are added, it is the most stable approach.

If we take this optimization into consideration, the φ function becomes the following:

We consider the analysis information for the current node as $AI = (Val, Con)$. The

$$\varphi_\ell = \begin{cases} (Val, Con) & \text{if } \ell \text{ does not modify state and does not impose or remove constraints} \\ (Val[x_i = [v, v]], Con) & \text{if } \ell \text{ modifies the variable } x_i \text{ with the value } v \\ (Val, Con \cup C_1) & \text{if } \ell \text{ imposes the constraint } C_1 \text{ and Lab(source)} < \text{Lab(destination)} \\ (Val, Con \setminus C_1) & \text{if } C_1 \text{ was enforced and } \ell \text{ invalidates it or if the scope of } C_1 \text{ ends} \end{cases}$$

The constraints are required to check whether the current state is satisfiable. We make use of an SMT solver to perform this check. Our solution is designed to support a variety of solvers in a plugin approach. At the time of writing, only integration with Z3 was implemented. This interaction is explained in further detail in the following sections of this paper.

To highlight how this analysis technique approximates the state of a program, we provide the following example:

```
function magicNumber(uint x) pure external returns(uint){
   uint index=0; //statement 1
   uint value=x; //statement 2
   require(x<15); //statement 3
   while(index<x) //statement 4
   {
        if(index\%2==0) //statement 5
        {
            value=value*2; //statement 6
        }
        else
        {
            value=value*3; //statement 7
        }
         x=x+1; //statement 8
   }
   return value; //statement 9
}
```

The approximations of state for each program point are displayed in Table 1.

Statements	*x*	*index*	*value*	*Constraints*
1	[0,∞]	∅	∅	{}
2	[0,∞]	[0,0]	∅	{}
3	[0,∞]	[0,0]	[0,∞]	{}
4	[0,∞]	[0,∞]	[0,∞]	{x<15}
5	[0,∞]	[0,∞]	[0,∞]	{x<15 && index<x}
6	[0,∞]	[0,∞]	[0,∞]	{x<15 && index<x && index%2==0}
7	[0,∞]	[0,∞]	[0,∞]	{x<15 && index<x && index%2!=0}
8	[0,∞]	[0,∞]	[0,∞]	{x<15}
9	[0,∞]	[0,∞]	[0,∞]	{x<15}
End	Data 9.2	[0,∞]	[0,∞]	{x<15}

Table 1: Interval analysis with constraints for the `magicNumber` function.

4 Slither

4.1 Overview

Slither[12] is a static analysis tool that offers security and good practices advice for smart contracts. Compared to other analysis tools, it runs remarkably fast. The tool obtains information by using the Solidity compiler and converting EVM bytecode into a proprietary intermediate representation called SlithIR. This new representation can be visualized using a printer, it also has an SSA (Static Single Assignment) variant. Before converting the initial code to SlithIR, it uses the information provided by the compiler to construct the CFG, Inheritance Graph. This is achieved by parsing the AST (Abstract Syntax Tree) resulting from contract compilation.

At the time of writing, Slither features 93 detectors[7]. These detectors are self-contained plugins that process the information compiled independently. Being an open-source tool, any user can contribute with new detectors. To do so, the contributor needs to extend an abstract class that defines the common behaviour among all detectors: processing contract information (i.e., detecting the bug); displaying the result; and attributes regarding the detector documentation (severity, confidence, etc).

In addition to detection purposes, the contract information can also be displayed using ‘printers‘. They display contract information internally gathered by Slither in a human-readable format. In the official documentation, the printers are divided into two categories: ‘Quick Review Printers‘ and ‘In-Depth Review Printers‘. The printers that we consider offer the most useful insights are the following: human-summary; contract-summary; loc (lines of code); cfg (control-flow-graph) and function summary.

Besides detecting issues and printing contract information, Slither also has many other capabilities. These features include: generating reports; generating code, extracting an interface from an existing contract; correcting code, either reformatting the contract, fixing simple vulnerabilities automatically or flattening the codebase; code checks, and checking compatibility with well-known standards or with newer features of Solidity.

The tool needs to be installed as a Python module via pip[8]. After the installation is performed, the user has access to the command line tool and the Python modules that can be included in a custom

[7]A detailed list is available on the Slither GitHub repository:`https://github.com/crytic/slither`

[8]`https://pypi.org/project/slither-analyzer/`

implementation. The GitHub page features a generous developer documentation[9]. The main entry point for a new implementation is a class called ‘Slither‘. An object of this type needs to be initialized by providing the path to the Solidity files that contain the targeted contract(s). After obtaining an instance of that class, the developer has access to many useful details that can be used to implement a new analysis tool or add a new detector to Slither. This information is presented as nested objects. The main object contains a list of contracts, each contract contains a list of functions, each function contains a list of nodes, and so on. Besides the children list, each object that models a contract element holds specific information, such as state variables for contracts and parameters for functions.

4.2 Limitations

To our best knowledge, Slither does not include the possibility of querying the contract state for each statement of a function, or at least at the start and end of execution. It does not implement interval analysis or other techniques capable of approximating the contract state. We consider that such a feature could be integrated into one of Slithers printers, aid in identifying code defects that are currently not detected, or improve the detection rate for issues that are already detected.

In addition to that, it appears that Slither does not take into consideration constraints imposed by previous statements. Constraints are added either by traditional conditional statements such as an `if` statement, or by instructions such as `assert` or `require`. Due to this, Slither is not always able to identify unreachable conditional branches. The unreachable branches could be an if/else block, the body of a loop with a contradiction acting as its condition, or even whole function sections if they are preceded by an `assert` or `require` statement that will never pass.

We present the following example:

```
pragma solidity 0.8.23;

contract BidContract {
    mapping(uint=>uint) public  bidders;
    function bid(uint bidderNumber) public  payable {
        require(msg.value>10);
        uint newBid=bidders[bidderNumber]+msg.value;
        if(newBid>10)
        {
            //Since msg.value>10 implies that newBid>10,
            //this brach will always execute
             bidders[bidderNumber]=newBid;
        }
        else
        {
            //Since the "then" branch is based on a tautology,
            //this branch will never execute
            revert("Inssuficient bid");
        }
  }
 }
```

Obs: Since we are working with unsigned integers, the variables cannot contain negative values.

If used on the contract above, Slither will signal some issues that are indeed present. The first issue relates to the fact that the version of Solidity that was used to compile it is too recent to be used. The other problem detected by Slither targets the absence of a function that allows the user to withdraw ether,

[9] Slither API documentation:`https://crytic.github.io/slither/slither.html`

we did not provide one in this contract because it is not relevant to our example. None of these defects are related to the problem that we present using this contract.

However, it will not address the fact that the `else` branch is unreachable.

Currently, our efforts are focused on this area. We aim to provide an implementation capable of approximating the program state as precisely as possible and identify issues based on the analysis information that we gather. In addition to that, we implemented a constraint system. Using a third-party solver, we can find code blocks that will never execute because they are on unreachable conditional branches.

Observation: We must also highlight an issue in Slithers parsing of the above contract that we discovered during our solutions development. If the `then` block of an `if` statement is empty and the `else` block is not, Slither will treat the latter as the "true" branch and the statement after the conditional statement as the "false" branch. To obtain a correct result, the truth values should be reversed. Bug encountered in **version 0.10.0** of the Slither pip package.

5 Solution architecture

Our architecture can be implemented using any language that offers support for the Object Oriented programming paradigm as long as the necessary third-party integrations are available. In the following presentation and diagrams, we use Slither as the contract parser and Z3 for the constraint solving component. These are the integrations that we implemented so far, but our architecture is easily extensible to other third-party solutions that fulfill similar roles. Currently, we are able to analyze the **SSA** (Static single-assignment) version of a program, as well as its **Non-SSA** form. We found the SSA variant to be more convenient since SlitherIR also has a SSA version.

In Figure 1, we present how our system receives input data, the interaction with the components that it depends on and how it produces the output data. The flow of execution is the following:

1. The user provides the path to a Solidity file along with the names of a contract and a function;
2. Use Slither to parse the contract;
3. Extract the information that it needs from the parsing result;
4. Run the least fixed point algorithm. Currently, only the worklist algorithm is implemented;
5. Translate constraints and program state into a representation compatible with the solver and call it;
6. Write the program state for each statement of the function into a file;
7. The user can manually analyze the program state generated by our solution.

Even though our solution currently only supports interval analysis and is integrated only with Z3 and Slither, we designed it to be extensible from the beginning. We achieved this by leveraging well-known software design principles and design patterns such as Adapter, Composite and Template Method. We make use of abstractions to define the general behavior, input and output that we expect a static analysis implementation to have. The same principle applies to external integrations, we define the data that we expect to extract from a contract parser. Figure 2 provides a diagram that contains the most important modules of our implementation. All modules currently contain an abstract definition as well as concrete implementations for Slither and Z3.

The labels, extremal labels and flow relation between nodes are computed in a similar fashion to the one described in Subsection 3.2. We determine the extremal labels and flow relation via graph search algorithms that run on the CFG data structure provided by Slither.

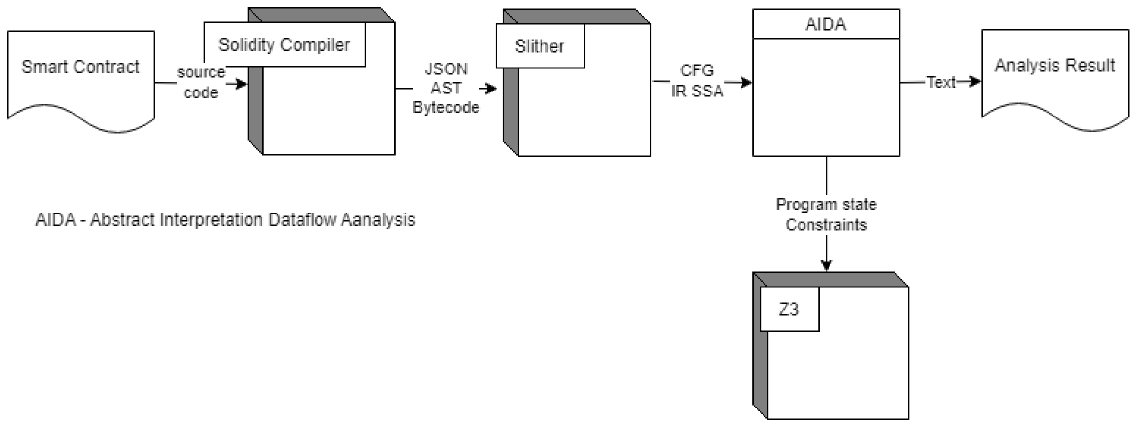


Figure 1: System context diagram of AIDA

We defined our own data structure to store information about a program variable. Along with the interval, we also store other useful information such as its name, type, scope, declaring node, if it is a reference type variable or not, etc. There are some special cases where some variables might require more information to be stored than others. For this case, we also defined an "additional_information" field as a dictionary. The volume of information in this field depends on the type of variable and its usage. For example, boolean variables that are used in constraints require more information about them to be stored for a proper interpretation of that constraint.

Since our domain does not satisfy ACC, we needed to implement widening operators. This is necessary to ensure termination when the program contains loops. Our implementation for the least upper bound for two simple numeric intervals is presented in the code snippet below:

```
def get_least_upper_bound_numeric_interval(first_element:
    NumericIntervalApproxValue, second_element: NumericIntervalApproxValue):
    if len(first_element) == 0:
        return second_element
    if len(second_element) == 0:
        return first_element
    lower_bound = first_element[0] if first_element[0] <= second_element[0] else
         float(
        "-inf")
    upper_bound = first_element[1] if first_element[1] >= second_element[1] else
         float(
        "inf")
    return NumericIntervalApproxValue((lower_bound, upper_bound))
```

Our implementation also supports complex types, such as arrays, mappings, and structures. Unlike scalar variables where the intervals can be simply represented as a pair of two numeric values, complex types require a different type of representation. Due to this, we modeled the intervals for complex types as dictionaries. This approach offers us a great deal of flexibility, we can use numerical indexes as keys for arrays or the field names for structs for example. For mappings, this representation comes naturally.

Even though our implementation supports nested complex types, these are not seen often. Since Ethereum requires a gas fee to run smart contracts, the code must be as resource efficient as possible, this is especially true for execution time and the amount of storage required. This mechanism heavily discourages developers from implementing complex scenarios and data structures

The way we designed our representation of program state for interval analysis is very similar to the

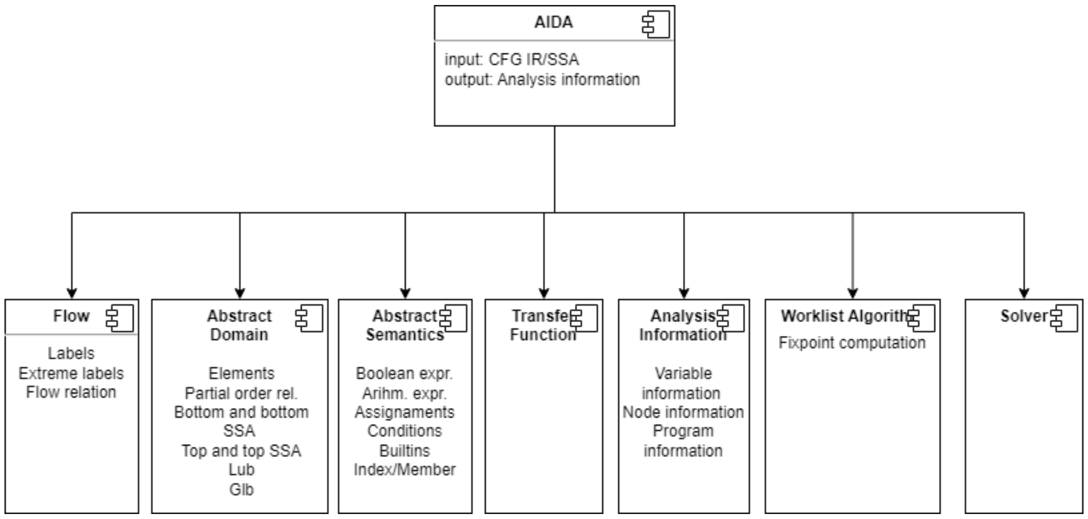


Figure 2: Modules of AIDA

one described in Section 2. The only difference lies in the fact that the information for a node does not contain a single collection of intervals corresponding to each variable. The information is distributed in the following categories: numeric variables, boolean variables and constraints. Grouping them together was not optimal for our use case since boolean variables and numeric variables have different intervals and properties that concern us during our analysis. This allows us to have a clear separation of arithmetic and boolean expressions. This matches perfectly with the way SlithIR represents conditional structures, the truth value of the condition is first stored in a temporary variable. After that, the 'CONDITION' SlithIR instruction is called and receives that temporary variable as its only parameter. Boolean variables that are subject to such calls have a special flag set in our representation of analysis information.

To integrate with Z3, we make use of the 'exec' builtin from Python. In this way, we can dynamically execute code that adds variables and constraints to the Z3 solver. Although the adapter implemented for Z3 will not work with any other solver without adjustments, only the code that is generated and executed dynamically needs to be changed. Due to our use of abstract classes and polymorphism, the current adapter can easily be changed with the newly implemented one. The flow of constraint checking will stay the same.

The flow of using a solver is the following:

1. Query analysis information to obtain all numeric variables that are present in boolean expressions with 'condition' flag set;
2. Query analysis information to obtain all boolean expressions that are marked as conditions;
3. Declare all numeric variables in solver;
4. Add interval constraints for all numeric variables that do not have the default interval for their type;
5. Add boolean expressions to the solver;
6. Check state satisfiability;
7. Decide if the current node is reachable or not from the current path.

An example of output that can be further investigated by the user is presented in the following sections.

To reach this state of implementation, we faced many challenges. Besides deciding the architecture and overall solution organization, being able to translate from the Slither representation to our representation and then to a representation that is compatible with the solver was particularly challenging. This required a considerable amount of empirical study of how Slither models different types of statements, variables, and data types. Due to the fact that we want our implementation to be easily extensible, we did not model our program state with any particular solver or other integration in mind. We chose the best overall approach to model program state. This makes it compatible with any integration, but extra steps must be taken to achieve it.

Another particularly challenging aspect was correctly identifying the constraints for each program point. Accumulating constraints as we iterate through the CFG seems like a natural approach. This solution is not semantically correct, since the CFG representation of a loop contains an edge from the end point of the loop to its start point. If we only accumulated constraints as we searched the CFG, that would have led to incorrect constraint representations for programs that contain loops.

We had to define a more complex constraint flow. Constraints can only flow downward in the control-flow graph. When processing an edge that has the end of a loop as it's source edge and the start of a loop as it's destination label, constraints are not carried over. We consider this to be the most stable approach since the start point of a loop is also reached from the initial flow of a program without the loop invariant.

6 Evaluating our solution

Our implementation is currently able to provide a state representation for each statement of a function. This representation is written in a text file and it is presented in a human-readable format. The constraint system that we implemented is currently the strong point of our implementation when compared to other static analysis tools. We evaluated our solution on a number of functions that contain arithmetic expressions, boolean expressions, loops, conditionals, and builtin functions (`require`, `assert`, `send`, `transfer`, `call`). A contract test suite, as well as an executable build of our tool will be available at `https://github.com/CDU55/FROM2024VulnerableSmartContracts`.

A good example that highlights the strength of our tool is the following

```
pragma solidity 0.8.23;

  contract DepositContract {

      mapping(address=>uint) public  deposits;
      function deposit() public payable {
          require(msg.value > 0);
          deposits[msg.sender]=deposits[msg.sender]+msg.value;
      }

      function withdraw() public  payable {
         require(deposits[msg.sender] > 0);
         payable(msg.sender).transfer(deposits[msg.sender]);
         //MISSING: set deposits[msg.sender] to 0
         //this assert statement will always revert
         //and the users cannot withdraw ether
         assert(deposits[msg.sender] == 0);
     }
}
```

We focus mainly on the ‘withdraw‘ function. Our constraint-based approach can correctly identify the fact that the endpoint of the function is not reachable. This is due to two contradictory constraints. Having both ‘deposits[msg.sender] > 0‘ as a constraint at the start of the contract means that the caller’s balance will always be positive, a usual condition for such functions. Since ‘deposits[msg.sender]‘ is not altered during execution, the constraint is still valid. When reaching the `assert` statement, the contract is led into an invalid state. The balance of a user cannot be greater than 0 and equal to 0 at the same time. This leads to a ‘Locked Ether‘ vulnerability case. Users can deposit currency and cannot withdraw it later.

The program state computed by our tool for the statement on line 15 in the example above is displayed in Figure 3.

```
Node 3:EXPRESSION assert(bool)(deposits[msg.sender] == 0)
 {
Numeric variables:{'block.timestamp': block.timestamp uint (1,inf)
, 'block.difficulty': block.difficulty uint (1,inf)
, 'block.number': block.number uint (1,inf)
, 'msg.sender': msg.sender uint (1,inf)
, 'msg.value': msg.value uint (1,inf)
, 'deposits': deposits:
},
Booleans variables:{'TMP_3': TMP_3 bool deposits[msg.sender] > 0 assert/require
},
Constraints:[('TMP_3', True)]
}
```

Figure 3: Example of state output for a program node

The summary provided by our tool highlights the fact that the constraint imposed by the assert statement contradicts an already existing constraint. This becomes clear when reviewing the code of the `withdraw` function along with the approximated program state.

Currently, our solution is not able to automatically signal issues, the only defect that it is able to automatically find is ‘Unreachable code‘. Thus, it can only be used as a helper tool for smart contract auditors at the moment. Automatic fault detection will be implemented in the future. The methodology that we implemented for evaluation was to run the tool, obtain the program state, and check if the targeted issue could easily be identified by the user when checking the output file.

Issues that could be identified by checking the output of our program are the following:

1. Array out of bounds
2. Division by zero
3. Unreachable code (Tautologies and contradictions in conditional instructions)
4. Missing validation for parameters and state variables

Table 2 displays how our tool performed compared to Slither on our contract collection. The contracts range from one-line functions that only contain the issue to more complex scenarios. If the issue can be reasonably deduced from the program state computed by our solution, then we consider it able to help in detecting that issue. We do not make any claim that our collection is an exhaustive one. However, these contracts highlight scenarios where Slither does not perform ideally.

Our solution	Slither	Contract	Function	Issue
✓	✗	BidContract	bid	Unreachable code
✓	✗	DepositContract	withdraw	Locked Ether
✓	✗	DivideByZeroMinimal	divide	Possible division by 0
✓	✗	ImproperDataValidation	participate	No parameter validation
✓	✗	OutOfBoundsArrayMinimal	getArrayElement	Array out of bounds
✓	✗	DivisionByZeroArray	getSomeResult	Possible division by 0

Table 2: The results of our evaluation

7 Current limitations

Even though we made significant progress towards a working solution, it still has limitations and we already theorized ways in which they could be mitigated. We have identified two categories of limitations: current implementation limitations and interval analysis limitations.

The first major limitation that our tool currently has is tied to the complexity of the contracts that it can process. Even though Solidity is smaller in the size of its instruction set than other well-known programming languages such as C, it still has plenty of features that must be interpreted. Since we implemented our own processors for the semantics of each instruction type, there is still work to be done. The common instructions among most programming languages are currently implemented, as well as the most used builtin functions and variables that are specific to Solidity. There is still work to do regarding other language-specific features such as ‘`abi.encode()`‘ and ‘`readInt8()`‘. This problem will be mitigated by adding semantics for elements of Solidity that are currently missing.

Another feature that is currently missing from our tool is interprocedural analysis. Out implementation that does not yet support calls between functions that are not Solidity builtins, meaning that it is limited to intraprocedural analysis. This issue will be mitigated by implementing a recursive approach to our current analysis algorithm.

We must also mention the limitations that interval analysis imposes. This is mostly related to the precision of approximation for the interval of each variable. If the intervals are not precise enough, an issue could only be signaled as "potentially present", not as "certainly present". Increasing precision by altering the $\top$ and $\bot$ values, which are used as the default values for variables could make the implementation more prone to mistakes. This issue could be mitigated by receiving more precise default intervals from the user via annotations or a configuration file. Another way that this issue could be mitigated is by adding more types of static analysis to help the already existing one.

8 Conclusions

In this progress report, we present our latest developments towards a new analysis tool that aims to find issues that are not currently detected by state-of-the-art tools. We provide a detailed presentation of how Slither runs and a set of instructions regarding how to build a new solution by leveraging the modules that it provides. We describe in detail how our tool is designed and how it interacts with third-party systems. Our implementation of interval analysis is the solution’s strong point currently, but it can easily be extended to other methods of static analysis and other third-party components. At the moment of writing, our tool can provide a summary of a program’s states. This summary allows the user to easily identify issues currently not detected by other similar tools, two such defects are described in this paper.

We also highlight our current limitations and present our plans to overcome them.

9 Bibliography

References

[1] *Remix IDE*. `https://remix-ide.readthedocs.io/en/latest/`.

[2] *Solhint — An open source project for linting Solidity code*. `https://protofire.github.io/solhint/`.

[3] Frances E. Allen (1970): *Control flow analysis*. In Robert S. Northcote, editor: *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, ACM, pp. 1–19, doi:10.1145/390013.808479.

[4] Frances E. Allen & John Cocke (1976): *A program data flow analysis procedure*. *Communications of the ACM* 19(3), p. 137, doi:10.1145/360018.360025.

[5] Andrei Arusoaie & Ștefan-Claudiu Susan (2024): *Towards Trusted Smart Contracts: A Comprehensive Test Suite For Vulnerability Detection*. *Empirical Software Engineering* 29(5), p. 117, doi:10.1007/s10664-024-10509-w.

[6] Bruno Blanchet (2002): *Introduction to abstract interpretation*.

[7] Vitalik Buterin et al. (2014): *A next-generation smart contract and decentralized application platform*. *white paper* 3(37), pp. 2–1.

[8] Agostino Cortesi & Matteo Zanioli (2011): *Widening and narrowing operators for abstract interpretation*. *Computer Languages, Systems & Structures* 37(1), pp. 24–42, doi:10.1016/j.cl.2010.09.001.

[9] Patrick Cousot (2021): *Dynamic interval analysis by abstract interpretation*, pp. 61–86. doi:10.1007/978-3-030-87348-6_4.

[10] Patrick Cousot & Radhia Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, Association for Computing Machinery, New York, NY, USA, p. 238–252, doi:10.1145/512950.512973.

[11] Dxo, Mate Soos, Zoe Paraskevopoulou, Martin Lundfall & Mikael Brockman (2024): *Hevm, a Fast Symbolic Execution Framework for EVM Bytecode*. In: *International Conference on Computer Aided Verification*, Springer, pp. 453–465, doi:10.1007/978-3-031-65627-9_22.

[12] Josselin Feist, Gustavo Grieco & Alex Groce (2019): *Slither: a static analysis framework for smart contracts*. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, pp. 8–15, doi:10.1109/WETSEB.2019.00008.

[13] Nicolas Halbwachs (1979): *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. Ph.D. thesis, Institut National Polytechnique de Grenoble-INPG; Université Joseph-Fourier ....

[14] Henk J. A. M. Heijmans (2020): *Complete lattices*. doi:10.1016/bs.aiep.2020.07.002.

[15] Uday Khedker, Amitabha Sanyal & Bageshri Sathe (2017): *Data flow analysis: theory and practice*. CRC Press, doi:10.1201/9780849332517.

[16] Gary A. Kildall (1973): *A unified approach to global program optimization*. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, Association for Computing Machinery, New York, NY, USA, p. 194–206, doi:10.1145/512927.512945.

[17] John McCarthy (1959): *A Basis for a Mathematical Theory of Computation*. In P. Braffort & D. Hirschberg, editors: *Computer Programming and Formal Systems*, *Studies in Logic and the Foundations of Mathematics* 26, Elsevier, pp. 33–70, doi:10.1016/S0049-237X(09)70099-0.

[18] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim & Marek Laskowski (2019): *Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack*. *Journal of Cases on Information Technology (JCIT)* 21(1), pp. 19–32, doi:10.4018/JCIT.2019010102.

[19] Satoshi Nakamoto (2008): *Bitcoin: A peer-to-peer electronic cash system*.

[20] Flemming Nielson, Hanne R Nielson & Chris Hankin (2015): *Principles of program analysis*. springer.

[21] Santiago Palladino: *The Parity Wallet Hack Explained*. `https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7`.

[22] Heidelinde Rameder, Monika Di Angelo & Gernot Salzer (2022): *Review of automated vulnerability analysis of smart contracts on Ethereum*. *Frontiers in Blockchain* 5, p. 814977, doi:10.3389/fbloc.2022.814977.s001.

[23] Grigore Roșu & Traian Florin Șerbănută (2010): *An overview of the K semantic framework*. *The Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[24] Michael I Schwartzbach (2008): *Lecture notes on static analysis*. *Basic Research in Computer Science, University of Aarhus, Denmark*.

[25] Nipun Sharma & Swati Sharma (2022): *A survey of Mythril, a smart contract security analysis tool for EVM bytecode*. *Indian J Natural Sci* 13, p. 75.

[26] Ştefan-Claudiu Susan & Andrei Arusoaie (2023): *Identifying Vulnerabilities in Smart Contracts using Interval Analysis*. *arXiv preprint arXiv:2309.13805*, doi:10.4204/EPTCS.389.12.

[27] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli & Martin Vechev (2018): *Securify: Practical security analysis of smart contracts*. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 67–82, doi:10.1145/3243734.3243780.