Fakultät für Informatik

Technische Universität München

Bachelor's thesis in Informatics

Algorithms for refinement of modal process rewrite systems

Philipp Meyer

Fakultät für Informatik

Technische Universität München

Bachelor's thesis in Informatics

Algorithms for refinement of modal process rewrite systems

Algorithmen zur Verfeinerung von modalen Prozessersetzungssystemen

Author:	Philipp Meyer
Supervisor:	Univ.-Prof. Dr. Dr. h.c. Javier Esparza
Advisor:	M. Sc. Jan Křetínský
Submission Date:	April 12, 2013

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, April 15th, 2013

Philipp Meyer

Contents

List of Figures

Listings

1 Introduction

As an extension to labelled transitions systems, modal transition systems (MTS) [LT88] have been widely used, especially in model checking. They provide a way to describe systems specifications in a way that allows stepwise refinement and composition of several refinements.

An MTS has two transitions, *may* transitions, which are admissible, and *must* transitions, which are necessary. A refinement of an MTS should then keep all still be able to perform all must transitions and all transitions it can perform are may transitions. This way one can gradually produce finer specifications that still conform to the original specification, until one arrives at a concrete implementation. Alternatively one can produce coarser specifications for abstraction.

There are many types of MTS, but formalisms to describe modal transitions systems with an infinite state space have only been explored recently [BK12]. For transitions systems, one powerful framework to describe them are process rewrite systems (PRS) [May00, Esp01]. They can be used to model many widely used systems such as pushdown automaton (PDA) or Petri nets (PN). By lifting PRS to the modal world, we obtain mPRS and their respective modal transitions systems such as mPDA or mPN.

Unfortunately, even for basic classes such as stateless PDA, known as binary process algebras (BPA), already simulation is undecidable [GH94]. As refinement is a generalization of both simulation and bisimulation, it is also undecidable on mBPAs and mPDAs.

However there is the subclass of visibly pushdown automata (vPDA) of PDAs, which is closed under all desirable operations and for which most problems are decidable [AM04]. They restrict the rules of a PDA by making visible which actions push a symbol on a stack and which pop one. They are still more expressive than finite automata and can be used to specify certain properties about the stack. For example, they have been applied to check parenthesis-like matching in XML streaming [KMV07] and checking pre/post-conditions for module calls in program analysis [AEM04].

Therefore we would like to use the modal view on vPDAs as well to allow modal specifcations and abstractions and would want to test refinement between these. Fortunately, not only simulation and bisimulation is decidable [Srb06], but also refinement [BK12]. Here we present an algorithm to decide the modal refinement on mvPDAs.

2 Theory

2.1 Processes

Processes are used to describe the state of many sequential or concurrent systems. The syntax here is taken from [May00] and [Esp01].

Definition 1 (Process)**.** The set of *processes* $\mathcal{P}$ over a set of constants $Const$ is given by

$$\frac{}{\varepsilon \in \mathcal{P}}\,(0) \qquad \frac{X \in Const}{X \in \mathcal{P}}\,(1) \qquad \frac{p \in \mathcal{P} \qquad q \in \mathcal{P}}{p \cdot q \in \mathcal{P}}\,(S) \qquad \frac{p \in \mathcal{P} \qquad q \in \mathcal{P}}{p \| q \in \mathcal{P}}\,(P)$$

where ε is the empty process, $X \in C$ are process constants, $\cdot$ means sequential composition and $\|$ means parallel composition.

Processes are considered equivalent under the smallest congruence relation such that the operator $\cdot$ is associative, $\|$ is associative and commutative and ε is a unit for both $\cdot$ and $\|$.

From here on we will denote processes by lowercase letters $p, q, \ldots$ and single process constants by uppercase letters $P, Q, \ldots$.

The class of processes that can be produced just with rule 1, i.e. contain no $\|$ or $\cdot$ and are not the empty process, is the class of *constant processes* **1**. The class of processes that can be produced just with rule 0, 1 and S, i.e. contain no $\|$, is the class of *sequential processes* **S**. The class of processes that can be produced just with rule 0, 1 and P, i.e. contain no $\cdot$, is the class of *parallel processes* **P**. The class of processes that can be produced by any combination of rules is the class of *general processes* **G**.

Definition 2 (Size of a process)**.** The *size* $|p|$ of a process p, is defined by

$$\begin{aligned} |\varepsilon| &= 0 \\ |X| &= 1 \\ |p \cdot q| &= |p| + |q| \\ |p\|q| &= |p| + |q| \end{aligned}$$

which is the number of constants appearing in the term.

2.2 Modal transition system

Modal transition systems are an extension of labeled transition systems with two types of transitions, may and must transitions. The intuition behind MTS is that in an refining system of an MTS, all must transitions have to be present, while the may transitions are allowed but not required.

Definition 3 (Modal transition system)**.** A *modal transition system (MTS)* over an action alphabet Act is a triple $(\mathcal{P}, \dashrightarrow, \longrightarrow)$ where $\mathcal{P}$ is a set of processes and $\longrightarrow \subseteq \dashrightarrow \subseteq \mathcal{P} \times Act \times \mathcal{P}$. An element $(p, a, q) \in \dashrightarrow$ is a *may transition*, written as $p \overset{\text{a}}{\dashrightarrow} q$, and an element $(p, a, q) \in \longrightarrow$ is a *must transition*, written as $p \xrightarrow{\text{a}} q$.

2.3 Modal refinement

For two processes of an MTS, we want to define modal refinement as a as an extension of bisimulation. One process refines another process if it a more specific version of a more general process. Both processes should match each others actions, except that the more general process can execute necessary must transitions and the more specific process can execute possible may transitions.

Out of simplicity, we regard only modal refinment for two processes from a single MTS. Modal refinement of processes from two different MTS can be reduced to this by taking the disjoint union of the MTS.

Definition 4 (Refinement)**.** Let $(\mathcal{P}, \dashrightarrow, \longrightarrow)$ be an MTS and $p, q \in \mathcal{P}$ be processes. We say that p *refines* q, written $p \leq_m q$, if there is a relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ such that $(p, q) \in \mathcal{R}$ and for every $(p, q) \in \mathcal{R}$ and every $a \in Act$:

1. If $p \overset{\text{a}}{\dashrightarrow} p'$ then there is a transition $q \overset{\text{a}}{\dashrightarrow} q'$ s.t. $(p', q') \in \mathcal{R}$.
2. If $q \xrightarrow{\text{a}} q'$ then there is a transition $p \xrightarrow{\text{a}} p'$ s.t. $(p', q') \in \mathcal{R}$.

Modal refinement can also be seen as a refinement game from a pair of processes (p, q) where each side plays an attacking transition and the other a defending transition to reach a new state. This type of game is often used to decide simulation or bisimulation [Srb06].

Thus if from the state (p, q) there is a transition $p \overset{\text{a}}{\dashrightarrow} p'$ or $q \overset{\text{a}}{\dashrightarrow} q'$, we will call this an *attacking transition* and a transition $q \overset{\text{a}}{\dashrightarrow} q'$ or $p \xrightarrow{\text{a}} p'$ from that state matching the type and action of the attacking transition a *defending transition*.

Then $p \leq_m q$ holds if there no winning strategy from (p, q), i.e. a sequence of attacking transitions such that for every choice of defending transition we will reach a state (p', q') from which there is an attacking transition but no defending transition.

2.4 Modal process rewrite system

A modal process rewrite system is a framework for defining a modal transition system by a finite set of rules. They are a straightforward extension of process rewrite systems, which can then model many transition systems such as pushdown automaton or Petri nets.

Definition 5 (Modal process rewrite system)**.** A *process rewrite system (PRS)* over an action alphabet Act is a finite relation $\Delta \subseteq \mathcal{P} \setminus \{\varepsilon\} \times Act \times \mathcal{P}$. Elements of Δ are called *rewrite rules.* A *modal process rewrite system (mPRS)* is a tuple $(\Delta_{\text{may}}, \Delta_{\text{must}})$ where $\Delta_{\text{may}}, \Delta_{\text{must}}$ are process rewrite systems such that $\Delta_{\text{may}} \subseteq \Delta_{\text{must}}$.

An mPRS $(\Delta_{\text{may}}, \Delta_{\text{must}})$ induces an MTS $(\mathcal{P}, \dashrightarrow, \longrightarrow)$ as follows:

$$\frac{(p, a, p') \in \Delta_{\text{may}}}{p \overset{\text{a}}{\dashrightarrow} p'}\ (1) \qquad \frac{(p, a, p') \in \Delta_{\text{must}}}{p \xrightarrow{\text{a}} p'}\ (2)$$

$$\frac{p \overset{\text{a}}{\dashrightarrow} p'}{p \cdot q \overset{\text{a}}{\dashrightarrow} p \cdot q}\ (3) \qquad \frac{p \xrightarrow{\text{a}} p'}{p \cdot q \xrightarrow{\text{a}} p' \cdot q}\ (4) \qquad \frac{p \overset{\text{a}}{\dashrightarrow} p'}{p \| q \overset{\text{a}}{\dashrightarrow} p \| q}\ (5) \qquad \frac{p \xrightarrow{\text{a}} p'}{p \| q \xrightarrow{\text{a}} p' \| q}\ (6)$$

2.5 Visibly pushdown automaton

A subclass of pushdown automata (PDA), visibly pushdown automata (vPDA) partition their action alphabet such that each action either is a call, internal or return transition.

Definition 6 (Visibly pushdown automaton)**.** A PRS Δ over the action alphabet Act is a *visibly pushdown automaton (vPDA)* if there is a partition $Act = Act_r \uplus Act_i \uplus Act_c$ such that every rule $(p, a, p') \in \Delta$ has the form

$$p = P \cdot S \qquad \text{and} \qquad p' = \begin{cases} Q & \text{if } a \in Act_r \quad \text{(return rule)} \\ Q \cdot T & \text{if } a \in Act_i \quad \text{(internal rule)} \\ Q \cdot T \cdot R & \text{if } a \in Act_c \quad \text{(call rule)} \end{cases}$$

for some $P, Q, S, T, R \in Const$. A *modal visibly pushdown automaton (mvPDA)* is an mPRS $(\Delta_{\text{may}}, \Delta_{\text{must}})$ such that Δ_{may} and Δ_{must} are vPDA.

We will have a look at the concepts introduced so far in an example.

Example 1 (mvPDA). *Suppose we want to create a specification for a vending machine selling coffee. It may accept any number of coins, but once it does, it nondeterministically chooses coffee or tea and must offer one beverage for each coin inserted. It may also offer the other beverage, but is not required to.*

An implementation of this specifiation wants to avoid nondeterminism and only chooses tea or coffee once the first choice is made. After that it only offers the chosen beverage until there are no more coins.

Figure 2.1 shows both implementation and specification modeled by an mvPDA. Note that may transitions are implied by the must transitions. The process $P \cdot S$ *is the initial process for the implementation, which stores the coin count as the number of symbols* M *on the stack and the beverage choice as the state* T *or* C*. The process* $Q \cdot S$ *is the initial process for the specification, which stores the count of coins as the number of* T *or* C *on the stack and the beverage choice as the stack symbol chosen.*

$$\begin{array}{rclcrcl}
P \cdot S & \xrightarrow{\text{coin}} & P \cdot M \cdot S & \qquad & Q \cdot S & \overset{\text{coin}}{\dashrightarrow} & Q \cdot T \cdot S \\
P \cdot M & \xrightarrow{\text{coin}} & P \cdot M \cdot M & & Q \cdot S & \overset{\text{coin}}{\dashrightarrow} & Q \cdot C \cdot S \\
C \cdot S & \xrightarrow{\text{coin}} & P \cdot M \cdot S & & Q \cdot T & \overset{\text{coin}}{\dashrightarrow} & Q \cdot T \cdot T \\
T \cdot S & \xrightarrow{\text{coin}} & P \cdot M \cdot S & & Q \cdot C & \overset{\text{coin}}{\dashrightarrow} & Q \cdot C \cdot C \\
P \cdot M & \xrightarrow{\text{tea}} & T & & Q \cdot T & \xrightarrow{\text{tea}} & Q \\
P \cdot M & \xrightarrow{\text{coffee}} & C & & Q \cdot T & \overset{\text{coffee}}{\dashrightarrow} & Q \\
T \cdot M & \xrightarrow{\text{tea}} & T & & Q \cdot C & \overset{\text{tea}}{\dashrightarrow} & Q \\
C \cdot M & \xrightarrow{\text{coffee}} & C & & Q \cdot C & \xrightarrow{\text{coffee}} & Q
\end{array}$$

Figure 2.1: Vending machine mvPDA

Example 2 (Refinement). *We can regard some refinement problems on the mvPDA from figure 2.1. For example, we can show* $T \cdot M \leq_m Q \cdot T$ *holds. From* $T \cdot M$ *the only may transition possible is* $T \cdot M \overset{tea}{\dashrightarrow} T$*, which is answered by* $Q \cdot T \overset{tea}{\dashrightarrow} Q$*. From* $Q \cdot T$ *the only must transition is* $Q \cdot T \xrightarrow{tea} Q$*, which is answered by* $T \cdot M \xrightarrow{tea} T$*. In both cases from the resulting state* (T, Q) *there are no more transitions,* $T \cdot M \leq_m Q \cdot T$*.*

On the other hand, $C \cdot M \leq_m Q \cdot T$ *does not hold, as from* $Q \cdot T$ *there is the transition* $Q \cdot T \overset{tea}{\dashrightarrow} T$*, but there is no transition of the form* $C \cdot M \overset{tea}{\dashrightarrow} p'$*.*

The main problem is to decide whether $P \cdot S \leq_m Q \cdot S$ holds. It is not easy to see if it does from the rules directly. However later we will construct a method to decide it algorithmically and show that it actually does not hold.

2.6 Attack tree

When regarding refinement as a game, we can represent the winning strategy in a tree. Here we will define attack trees as a representation of partially or fully explored strategies, which can then be used to decide refinement.

Definition 7 (Attack tree)**.** An *attack tree* over a set of processes $\mathcal{P}$ is a rooted tree where each node has two kinds of children. It is given by a triple $((p,q),O,C)$, representing the tree with the root node labeled by $(p,q) \in \mathcal{P}^2$, the set of open edges O leading to states $(p',q') \in \mathcal{P}^2$ and the set of closed edges C leading to the attack trees that are children of the root node.

For an attack tree $T = ((p,q),O,C)$, we will use the short notations $T_r = (p,q)$ for the root, $T_O = O$ for the set of states open edges lead to and $T_C = C$ for the set of child trees closed edges lead to.

The set of attack trees $\mathcal{T}$ constructable from an MTS $(\mathcal{P}, \dashrightarrow, \longrightarrow)$ are defined inductively by:

$$\frac{p,q \in \mathcal{P}, p \overset{a}{\dashrightarrow} p'}{((p,q),\{(p',q') \mid q \overset{a}{\dashrightarrow} q'\},\emptyset) \in \mathcal{T}} \quad (1)$$

$$\frac{p,q \in \mathcal{P}, q \xrightarrow{a} q'}{((p,q),\{(p',q') \mid p \xrightarrow{a} q'\},\emptyset) \in \mathcal{T}} \quad (2)$$

$$\frac{T \in \mathcal{T} \qquad R \in \mathcal{T} \qquad R_r \in T_O}{(T_r, T_O \setminus \{R_r\}, T_C \cup \{R\}) \in \mathcal{T}} \quad (3)$$

Rules 1 and 2 specify an initial tree for an attacking transition with edges to states for each possible defensive transitions, while rule 3 replaces an open edge to a state with a tree with that state as its root.

As we can see from the construction rules, every tree has a corresponding attacking transition from the root node, while for each defending transition applicable from that state and attacking transition there is an edge to either an open state or a child tree. Therefore we can identify nodes with attacking transitions and edges with defending transitions.

Any attack tree has finite depth, so wan define the set of all *subtrees* of T, including T itself, recursively by $subtree(T) = T \cup \left(\bigcup_{T' \in T_C} subtree(T')\right)$.

The set of all *open states* of T are the states (p', q') that have an open edge to it, that is $open(T) = \bigcup_{T' \in subtree(T)} T'_O$ or equivalently $open(T) = T_O \cup \left(\bigcup_{T' \in T_C} open(T')\right)$.

We say that a tree is *closed* if it has no open states, that is $closed(T) \iff open(T) = \emptyset$.

The construction rules for attack trees only allow us to add a tree to the root node as a subtree if there is an open edge. However, the following lemma shows that we can replace any open edge in the tree by an appropriate tree.

Lemma 1 (Tree composition)**.** *If there are attack trees T and R with $R_r \in open(T)$, then there is an attack tree S with $S_r = T_r$ and $open(S) = open(T) \setminus \{R_r\} \cup open(R)$. and $s \in T_O$*

Proof. We prove the proposition by induction on the number of proper subtrees with an open edge to R_r, that is $n = |\{T' \in subtree(T) \mid T' \neq T \wedge R_r \in open(T')\}|$:

1. $n = 0$: Then $R_r \in T_O$ and $R_r \notin open(T')$ for $T' \in T_C$, so with rule 3 we can construct $S = (T_r, T_O \setminus \{R_r\}, T_C \cup \{R\})$ with $open(S) = open(T) \setminus \{R_r\} \cup open(R)$

2. $n \geq 1$: Then there is $T' \in T_C$ such that $R_r \in open(T')$. T' at least does not have itself as a proper subtree with an open edge to R_r, so we can apply the induction hypothesis to obtain S' with $S'_r = T'_r$ and $open(S') = open(T') \setminus \{R_r\} \cup open(R)$

   As T' was added to T_C some point in the construction of T, we can substitute T' with S' and obtain T'' with $T''_r = T_r$, $T''_O = T_O$ and $T''_C = T_C \setminus \{T'\} \cup \{S'\}$. We have $open(T'') = T_O \cup \left(\bigcup_{R' \in T_C \setminus \{T'\}} open(R')\right) \cup open(S')$.

   If $R_r \neq open(T'')$, then $open(T'') = open(T) \setminus \{R_r\} \cup open(R)$ and we are done. Otherwise T'' has less subtrees with an open edge to R_r, therefore we can apply the induction hypothesis on it to obtain S with $S_r = T_r$ and $open(S) = open(T'') \setminus \{R_r\} \cup open(R) = open(T) \setminus \{R_r\} \cup open(R)$.

□

The following theorem gives us the equivalence of closed trees and non-refining processes. With that result, we can use the attack tree structure to argue over refinement instead of the refinement relation.

Theorem 1 (Attack tree refinement). *For an MTS $(\mathcal{P}, \dashrightarrow, \longrightarrow)$ and processes $p, q \in \mathcal{P}$:*

$$(p \leq_m q) \iff \neg\exists T \in \mathcal{T} : T_r = (p, q) \wedge closed(T)$$

Proof. $\Rightarrow$: Assume $p \leq_m q$. Then there is a refinement relation $\mathcal{R}$. To show that for $(p, q) \in \mathcal{R}$ there is no closed tree from (p, q), we show the contraposition that for any $T \in \mathcal{T}$, if T is closed, then $T_r \notin R$.

Recall that for any T there is an attacking transition from T_r and the edges correspond to the appropriate defending transition. Further if T is closed, we have $T_O = \emptyset$ and every $T' \in T_C$ is also closed.

Now we show the contraposition by induction over the number of subtrees of T, that is $n = |subtrees(T)|$:

1. $n = 1$: Then there is an attacking transition and as $T_C = \emptyset$ there is no defending transition, therefore $(p, q) \notin \mathcal{R}$.

2. $n > 1$: Then there is an attacking transition and for every defending transition leading to (p', q'), there is an edge to a closed tree T' with $T'_r = (p', q')$. T' is a proper subtree of T and has less subtrees itself, so by induction hypothesis we have $(p', q') \notin \mathcal{R}$ and therefore $(p, q) \notin \mathcal{R}$.

$\Leftarrow$: Assume that there is no closed attack tree T with $T_r = (p, q)$. To show $p \leq_m q$, we show that $\mathcal{R} := \{(p', q') \mid \neg\exists T : T_r = (p', q') \wedge closed(T)\}$ is a valid refinement relation with $(p, q) \in \mathcal{R}$.

For any attacking transition and $(p, q) \in \mathcal{R}$, by inference rule 1 or 2 there exists an attacking tree T with $T_r = (p, q)$. From all such T, choose one where $open(T)$ is minimal with regard to the inclusion order. There exists $(p', q') \in open(T)$ with $(p', q') \in \mathcal{R}$, because otherwise there would be a closed attack tree T' with $T'_r = (p', q')$ and with lemma 1 we would get T'' with $T''_r = T_r$ and $open(T'') = open(T) \setminus \{(p', q')\} \subsetneq open(T)$ in contradiction to the minimality of $open(T)$. So for the attacking transition from (p, q) there is a defending transition to (p', q') with $(p', q') \in \mathcal{R}$. □

Example 3. *For the MTS induced by the vending machine mvPDA from figure 2.1, attack trees for some states are displayed in figure 2.2. Tree nodes are displayed in rectangles with their associated attacking transition below them, edges are labeled with their defending transitions and open states are shown in rectangles with rounded corners.*

These trees can be combined to form a closed tree shown in figure 2.3. With theorem 1, this shows that $P \cdot S \leq_m Q \cdot S$ does not hold. A winning strategy can be read of the tree.

$$\begin{tikzcd}
T_1: & \boxed{(P \cdot S, Q \cdot S)} \arrow[r, "Q \cdot S \overset{\text{coin}}{\dashrightarrow} Q \cdot T \cdot S"] \arrow[dr, "Q \cdot S \overset{\text{coin}}{\dashrightarrow} Q \cdot C \cdot S"] & (P \cdot M \cdot S, Q \cdot T \cdot S) \\
& P \cdot S \overset{\text{coin}}{\dashrightarrow} P \cdot M \cdot S & (P \cdot M \cdot S, Q \cdot C \cdot S)
\end{tikzcd}$$

$$\begin{tikzcd}
T_2: & \boxed{(P \cdot M \cdot S, Q \cdot T \cdot S)} \arrow[r, "Q \cdot T \overset{\text{coin}}{\dashrightarrow} Q \cdot T \cdot T"] & (P \cdot M \cdot M \cdot S, Q \cdot T \cdot T \cdot S) \\
& P \cdot M \overset{\text{coin}}{\dashrightarrow} P \cdot M \cdot M &
\end{tikzcd}$$

$$\begin{tikzcd}
T_3: & \boxed{(P \cdot M \cdot S, Q \cdot C \cdot S)} \arrow[r, "Q \cdot C \overset{\text{coin}}{\dashrightarrow} Q \cdot C \cdot C"] & (P \cdot M \cdot M \cdot S, Q \cdot C \cdot C \cdot S) \\
& P \cdot M \overset{\text{coin}}{\dashrightarrow} P \cdot M \cdot M &
\end{tikzcd}$$

$$\begin{tikzcd}
T_4: & \boxed{(P \cdot M \cdot M \cdot S, Q \cdot T \cdot T \cdot S)} \arrow[r, "Q \cdot T \overset{\text{coffee}}{\dashrightarrow} Q"] & (T \cdot M \cdot S, Q \cdot T \cdot S) \\
& P \cdot M \overset{\text{coffee}}{\dashrightarrow} T &
\end{tikzcd}$$

$$\begin{tikzcd}
T_5: & \boxed{(P \cdot M \cdot M \cdot S, Q \cdot C \cdot C \cdot S)} \arrow[r, "Q \cdot C \overset{\text{tea}}{\dashrightarrow} Q"] & (T \cdot M \cdot S, Q \cdot C \cdot S) \\
& P \cdot M \overset{\text{tea}}{\dashrightarrow} T &
\end{tikzcd}$$

$$\begin{tikzcd}
T_6: & \boxed{(T \cdot M \cdot S, Q \cdot T \cdot S)} & T_7: & \boxed{(T \cdot M \cdot S, Q \cdot C \cdot S)} \\
& Q \cdot C \xrightarrow{\text{tea}} Q & & Q \cdot C \xrightarrow{\text{coffee}} Q
\end{tikzcd}$$

Figure 2.2: Initial attack trees for the vending machine MTS

2.7 Attack rule

The attack trees we represent strategies for any MTS. However, their states can be arbitrarily large and there can is an infinite number of possible trees starting at a certain state. We would to define a similiar concept for mvPDA, called attack rules, which can be used to represent parts of an attack tree for the corresponding MTS.

Definition 8 (Attack rule)**.** An *attack rule* is a tuple $((p, q), S)$ with $p, q \in \mathcal{P}$ and $S \subseteq \mathcal{P}$. It is written as $(p, q) \longrightarrow_a S$

$$\begin{tikzcd}
 & T_2 \arrow[r] & T_4 \arrow[r] & T_6 \\
T_1 \arrow[ur] \arrow[dr] & & & \\
 & T_3 \arrow[r] & T_5 \arrow[r] & T_7
\end{tikzcd}$$

Figure 2.3: Combined attack tree for the vending machine MTS

For an mvPDA $(\Delta_{\text{may}}, \Delta_{\text{must}})$, the attack rules obtainable from the rewrite rules are given by:

$$\frac{(p, a, p') \in \Delta_{\text{may}}}{(p, q) \longrightarrow_a \{(p', q') \mid (q, a, q') \in \Delta_{\text{may}}\}} \, (1)$$

$$\frac{(q, a, q') \in \Delta_{\text{must}}}{(p, q) \longrightarrow_a \{(p', q') \mid (p, a, p') \in \Delta_{\text{must}}\}} \, (2)$$

$$\frac{(p, q) \longrightarrow_a S \uplus \{(p', q')\} \qquad (p', q') \longrightarrow_a S' \qquad \forall (p'', q'') \in S' : |p''| = 1}{(p, q) \longrightarrow_a S \cup S'} \, (3)$$

$$\frac{(p, q) \longrightarrow_a S \uplus \{(p' \cdot P, q' \cdot Q)\} \qquad (p', q') \longrightarrow_a S' \qquad \forall (p'', q'') \in S' : |p''| = 1}{(p, q) \longrightarrow_a S \cup \{(p'' \cdot P, q'' \cdot Q) \mid (p'', q'') \in S'\}} \, (4)$$

Due to the constraints on the rewrite rules of an mvPDA and the construction of the attack rules, we can see that for any rule $(p, q) \longrightarrow_a S$, it holds that $|p| = |q| = 2$ and for all $(p', q') \in S$ that $1 \leq |p'| = |q'| \leq 3$.

When the rules 3 and 4 combine a rule $(p, q) \longrightarrow_a S \uplus \{(p', q')\}$ and a rule $(p', q') \longrightarrow_a S'$ on right, it always holds that $|p'| = 2$ or $|p'| = 3$ and for all $(p'', q'') \in S'$ $|p''| = 1$. We will call a rule $p \longrightarrow_a S$ a *right-hand side* rule if $\forall (p', q') \in S : |p'| = 1$ and otherwise a *left-hand side* rule. This partitions the set of rules into two classes.

As the number of rules for an mvPDA is finite and all attack rules produce states with processes of a bounded size, we see that the set of all attack rules is finite. We will use this and the inductive nature of the rules to develop the algorithm to decide refinement later.

First we need to prove that attack rules can be used to obtain attack trees. As rewrite rules and attack rules only consist of processes of fixed size and transitions and attack trees consist of processes of arbitrary size, we need to have a mapping between them. This is possible for mvPDA, as they only defines rules from sequential processes with a fixed size, and the MTS rules only induce bigger processes.

Lemma 2. *Given an MTS generated by a mvPDA, for $|p| \geq 2$, $|q| \geq 2$, p, q sequential and any $s, t \in \mathcal{P}$:*

$$p \overset{a}{\dashrightarrow} p' \iff p \cdot s \overset{a}{\dashrightarrow} p' \cdot s \qquad \text{and} \qquad q \xrightarrow{a} q' \iff q \cdot t \xrightarrow{a} q' \cdot t$$

Proof. $\Rightarrow$: Follows directly from the induction rules of an MTS from an mPRS.

$\Leftarrow$: In the inference chain for $p \cdot s \overset{\text{a}}{\dashrightarrow} p' \cdot s$, there is a $(r, a, r') \in \Delta_{\text{may}}$ which was used to obtain that rule with $p \cdot s = r \cdot s'$ and $p' \cdot s = r' \cdot s'$. As $|r| \leq |p|$, $|r'| \leq |p'$ and p, p', r, r' are all sequential, there is s'' with $p = r \cdot s''$ and $p' = r \cdot s''$. Then we can infer the transition $r \cdot s'' \overset{\text{a}}{\dashrightarrow} r' \cdot s'' = p \overset{\text{a}}{\dashrightarrow} p'$. □

Consequently, we can apply this to attack trees to extend their states with processes.

Lemma 3. *Given an MTS generated by a mvPDA and any $s, t \in \mathcal{P}$:*

If there is an attack tree T with $T_r = (p, q)$, then there is an attack tree R with $R_r = (p{\cdot}s, q{\cdot}t)$ and $open(R) = \{(p' \cdot s, q' \cdot t) \mid (p', q') \in open(T)\}$.

Proof. If we have a tree T, we can apply lemma 2 on each rule which generated the tree and obtain R. □

While our attack rules are not powerful enough to represent any attack tree, they can represent certain parts. A part of a tree is essentially a node with all edges and nodes to a set of ancestors, while a partition is a disjunct union of parts resulting in the complete tree.

Definition 9 (Partition of an attack tree)**.** A partition P of an attack tree T is given by a set of subtrees $P \subseteq subtree(T)$ with $T \in P$.

For $R_1, R_2 \in P$, we define a partial ordering $R_1 \leq R_2 \iff R_1 \in subtree(R_2)$ and consequently $R_1 < R_2 \iff R_1 \leq R_2 \wedge R_1 \neq R_2$. We define the partition successors of $R \in P$ given P as $succ_P(R) = \{R' \in P \mid R' < R \wedge \neg\exists R'' : R' < R'' \wedge R'' < R\}$.

A rule should than correspond to a part, or represent it, if it can be extended such that it leads from the root node of the part to all its successors.

Definition 10 (Part represented by an attack rule)**.** A subtree $R \in P$ in a partition is said to be *represented* by an attack rule $(p, q) \longrightarrow_a S$ if there exist $s, t \in \mathcal{P}$ such that $T_r = (p \cdot s, q \cdot t)$ and $\{R'_r \mid R' \in succ_P(R)\} = \{(p' \cdot s, q' \cdot t) \mid (p', q') \in S\}$

Now we can prove our main theorem, which states that there is an attack rule leading to the empty set for every closed tree.

Theorem 2. *For an mvPDA* $(\Delta_{may}, \Delta_{must})$ *with its induced MTS* $(\mathcal{P}, \dashrightarrow, \longrightarrow)$*, it holds that for any* $P, S, Q, R \in Const$*:*

$$\exists T : T_r = (P \cdot S, Q \cdot R) \wedge closed(T) \iff (P \cdot S, Q \cdot T) \longrightarrow_a \emptyset$$

Proof. $\Rightarrow$: Assume T to be closed tree with $T_r = (P \cdot S, Q \cdot R)$.

First we show that if there is a partition $P = \{T'_1, ..., T'_n\}$ such that each part is represented by an attack rule, then there is an attack rule $(P \cdot S, Q \cdot T) \longrightarrow_a \emptyset$ This is shown by induction on n:

1. $n = 1$: Then $P = \{T\}$ and there is a rule $(p, q) \longrightarrow_a S$ representing T. As $(p \cdot s, q \cdot t) = T_r = (P \cdot T, Q \cdot R)$ and $|p| = |q| = 2$, we have $(p, q) = (P \cdot T, Q \cdot R)$ and as $succ_P(T) = \emptyset$ we have $S = \emptyset$. Then the rule is $(P \cdot T, Q \cdot R) \longrightarrow_a \emptyset$.

2. $n > 1$: Let T' be the subtree with $T'_r = (P \cdot S, Q \cdot R)$ As $n > 1$, there is $T'' \in succ_P(T')$ where $T''_r = (p', q')$. Let $a = (P \cdot S, Q \cdot R) \longrightarrow_a S$ be the representing rule of T'. We have $(p', q') \in S$ and necessarily $|p'| = |q'| \geq 2$, as otherwise there would be no rule applicable from that state and therefore T'' would not exist. So a is a left-hand side rule.

   For every subtree $T'' \in P$ with $succ_P(T'') = \emptyset$, we have for the representing rule $b = (p, q) \rightarrow \emptyset$, so that is a right-hand side rule. Every path in T eventually leads to such a subtree.

   Then by following the children of the subtrees from T', we will eventually come to a subtree T' succeded by a subtree T'' such that the rule represeenting T' is a left-hand side rule and the rule representing T'' is a right-hand side rule.

   The partition $P' = P \setminus \{T'\}$ then is again a partition of T where $succ_{P'}(T') = succ_P(T') \setminus \{T''\} \cup succ_P(T'')$ and other successors are unchanged. We now show that we can construct a rule representing T' in P':

   Let $a_1 = (p, q) \longrightarrow_a S$ be the rule representing T' and $a_2 = (p', q') \longrightarrow_a S'$ be the rule representing T''. For a_1 for T' we have that there is $s, t \in \mathcal{P}$ and $(p'', q'') \in S$

with $T''_r = (p'' \cdot s, q'' \cdot t)$. For a_2 we have that there is $s', t' \in \mathcal{P}$ with $T''_r = (p' \cdot s', q' \cdot t')$.

Then $(p'' \cdot s, q'' \cdot t) = (p' \cdot s', q' \cdot t')$. As $2 \leq |p''| = |q''| \leq 3$ and $|p'| = |q'| = 2$ either $s = s'$ and $t = t'$ or $P \cdot s = s'$ and $Q \cdot t = t'$ for some $P, Q \in Const$.

In the first case, we have $(p', q') = (p'', q'')$ and we can apply rule 3 to obtain $(p, q) \longrightarrow_a S \setminus \{(p'', q'')\} \cup S'$. With $\{(p' \cdot s, q' \cdot t) \mid (p', q') \in S \setminus \{(p'', q'')\} \cup S'\} = \{T'_r \mid succ_{P'}(T')\}$, it represents T' in P'.

In the second case, we have $(p' \cdot P, q' \cdot Q) = (p'', q'')$ and we can apply rule 4 to obtain $(p, q) \longrightarrow_a S \setminus \{(p'', q'')\} \cup \{(p'' \cdot P, q'' \cdot Q) \mid (p'', q'') \in S'\}$. With $\{(p' \cdot s, q' \cdot t) \mid (p', q') \in S \setminus \{(p'', q'')\}\} \cup \{(p'' \cdot P \cdot s, q'' \cdot Q \cdot t) \mid (p'', q'') \in S'\} = \{T'_r \mid succ_{P'}(T')\}$, it represents T' in P'.

Then as P' is a partition for T having a rule representing each part with $n - 1$ elements, we can apply the induction hypothesis and obtain the rule is $(P \cdot T, Q \cdot R) \longrightarrow_a \emptyset$.

Now we need to show there is an initial partition for T represented by attack rules. If we initially take $P = subtrees(T)$, for each $T' \in P$ we have: There is an attacking transition from T'_r which induced R. As $succ_P(R) = T'_C$, for each $T'' \in succ_P(T')$ there is an appropriate defending transition to T''_r and as $T'_O = \emptyset$ for each defending transition a $T'' \in succ_P(T')$

Let $T'_t = (p \cdot s, q \cdot t)$ with $|p| = |q| = 2$. By lemma 2, for each transition $p \cdot s \overset{a}{\dashrightarrow} p' \cdot s$ there is an inducing $(p, a, p') \in \Delta_{\text{may}}$ and for each $q \cdot t \overset{a}{\dashrightarrow} q' \cdot t$ there is an inducing $(q, a, q') \in \Delta_{\text{may}}$. The same holds for $\xrightarrow{a}$ and Δ_{must}. So there is a rule $(p, q) \longrightarrow_a \{(p', q') \| (q, a, q') \in \Delta_{\text{may}}\}$ which represents T'.

$\Leftarrow$: We show that if $(p, q) \longrightarrow_a S$, then there is a tree T with $T_r = (p, q)$ such that $open(T) = S$ by induction on the construction of $(p, q) \longrightarrow_a S$:

1. It was constructed by rule 1 from $(p, a, p') \in \Delta_{\text{may}}$. Then there is an attacking transition $p \overset{a}{\dashrightarrow} p'$ and for every $(q, a, q') \in \Delta_{\text{may}}$ there is an induced defending transition $q \overset{a}{\dashrightarrow} q'$. Then $S = \{(p', q') | q \overset{a}{\dashrightarrow} q'\}$ and by attack tree inference rule 1 there is $T = ((p, q), S, \emptyset)$ with $open(T) = S$.

2. It was constructed by rule 2 from $(q, a, q') \in \Delta_{\text{must}}$. Then there is an attacking transition $q \xrightarrow{a} q'$ and for every $(p, a, p') \in \Delta_{\text{may}}$ there is an induced defending transition $p \xrightarrow{a} p'$. Then $S = \{(p', q') | p \overset{a}{\dashrightarrow} p'\}$ and by attack tree inference rule 2 there is $T = ((p, q), S, \emptyset)$ with $open(T) = S$.

3. It was constructed by rule 3 from $(p, q) \longrightarrow_a S'' \uplus \{(p', q')\}$ and $(p', q') \longrightarrow_a S'$ with $S = S'' \cup S'$. Then by induction hypothesis there is a tree T' with $T'_r = (p', q')$ and $open(T') = S'$ and a tree T'' with $T''_r = (p, q)$ and $open(T'') = S'' \uplus \{(p', q')\}$. By applying lemma 1 on T' and T'' there is a tree T with $T_r = (p, q)$ with $open(T) = S'' \cup S' = S$.

4. It was constructed by rule 4 from $(p, q) \longrightarrow_a S'' \uplus \{(p' \cdot P, q' \cdot Q)\}$ and $(p', q') \longrightarrow_a S'$ with $S = S'' \cup S'''$ and $S''' = \{(p'' \cdot P, q'' \cdot Q) \mid (p'', q'') \in S'\}$. Then by induction hypothesis there is a tree T' with $T'_r = (p', q')$ and $open(T') = S'$ and a tree T'' with $T''_r = (p, q)$ and $open(T'') = S'' \uplus \{(p' \cdot P, q' \cdot Q)\}$. By applying lemma 3 on T' there is a tree T''' with $T'''_r = (p' \cdot P, q' \cdot Q)$, $open(T''') = O''' \uplus \{(p' \cdot P, q' \cdot Q)\}$ and $O''' = \{(p'' \cdot P, q'' \cdot Q) \mid (p'', q'') \in S'\} = S'''$. By applying lemma 1 on T'' and T''' there is a tree T with $T_r = (p, q)$ and $open(T) = S'' \cup S''' = S$.

Therefore if $(P \cdot S, Q \cdot R) \longrightarrow_a \emptyset$, then there is a tree T with $T_r = (P \cdot S, Q \cdot R)$ and $open(T) = \emptyset$. □

$$T_1 \quad \boxed{(P \cdot S, Q \cdot S) \longrightarrow_a \{(P \cdot M \cdot S, Q \cdot C \cdot S), (P \cdot M \cdot S, Q \cdot T \cdot S)\}}$$

$$T_2 \quad \boxed{(P \cdot M, Q \cdot T) \longrightarrow_a \{(P \cdot M \cdot M, Q \cdot T \cdot T)\}}$$

$$T_3 \quad \boxed{(P \cdot M, Q \cdot C) \longrightarrow_a \{(P \cdot M \cdot M, Q \cdot C \cdot C)\}}$$

$$T_4 \quad \boxed{(P \cdot M, Q \cdot T) \longrightarrow_a \{(C, Q)\}}$$

$$T_5 \quad \boxed{(P \cdot M, Q \cdot C) \longrightarrow_a \{(T, Q)\}}$$

$$T_6 \quad \boxed{(C \cdot M, Q \cdot T) \longrightarrow_a \emptyset}$$

$$T_7 \quad \boxed{(T \cdot M, Q \cdot C) \longrightarrow_a \emptyset}$$

Figure 2.4: Attack rules for partition 1 of vending machine tree

Example 4. *Again we regard the vending machine mvPDA from figure 2.1. We will see how to derive the attack tree from figure 2.3 with attack rules to prove that $P \cdot S \leq_m Q \cdot S$ does not hold.*

Initially we take the finest partition of the attack tree, where every node has to a basic attack rules representing it, as shown in figure 2.4. Note that the rules from T_1, T_2, and T_3 are left-hand side rules and the rules from T_4, T_5, T_6, T_7 are right-hand side rules. So the only we can combine are the ones from T_2 with T_4 and T_3 with T_5. Then we obtain the partition and rules shown in figure 2.5.

After that we can combine the rules from T_2 with T_6 and from T_3 with T_7. This results in the rules shown in figure 2.6. Finally we combine T_1 first with T_2 and then with T_3 to obtain the

$$\begin{tikzcd}
T_1 & & & \boxed{(P\cdot S, Q\cdot S) \longrightarrow_a \{(P\cdot M\cdot S, Q\cdot C\cdot S), (P\cdot M\cdot S, Q\cdot T\cdot S)\}} \\
T_2 \arrow[r] & T_4 & & \boxed{(P\cdot M, Q\cdot T) \longrightarrow_a \{(C\cdot M, Q\cdot T)\}} \\
T_3 \arrow[r] & T_5 & & \boxed{(P\cdot M, Q\cdot C) \longrightarrow_a \{(T\cdot M, Q\cdot C)\}} \\
T_6 & & & \boxed{(C\cdot M, Q\cdot T) \longrightarrow_a \emptyset} \\
T_7 & & & \boxed{(T\cdot M, Q\cdot C) \longrightarrow_a \emptyset}
\end{tikzcd}$$

Figure 2.5: Attack rules for partition 2 of vending machine tree

$$\begin{tikzcd}
T_1 & & & \boxed{(P\cdot S, Q\cdot S) \longrightarrow_a \{(P\cdot M\cdot S, Q\cdot C\cdot S), (P\cdot M\cdot S, Q\cdot T\cdot S)\}} \\
T_2 \arrow[r] & T_4 \arrow[r] & T_6 & \boxed{(P\cdot M, Q\cdot T) \longrightarrow_a \emptyset\}} \\
T_3 \arrow[r] & T_5 \arrow[r] & T_7 & \boxed{(P\cdot M, Q\cdot C) \longrightarrow_a \emptyset}
\end{tikzcd}$$

Figure 2.6: Attack rules for partition 3 of vending machine tree

rule shown in figure 2.5, which represents the whole tree. With this we can decide that there is a winning strategy from $(P \cdot S, Q \cdot S)$, therefore the states do not refine.

$$\begin{tikzcd}
 & T_2 \arrow[r] & T_4 \arrow[r] & T_6 & \\
T_1 \arrow[ur] \arrow[dr] & & & & \boxed{(P\cdot S, Q\cdot S) \longrightarrow_a \emptyset} \\
 & T_3 \arrow[r] & T_5 \arrow[r] & T_7 &
\end{tikzcd}$$

Figure 2.7: Attack rules for partition 4 of vending machine tree

3 The refinement algorithm

3.1 Description

The given theory then lead to the development of a program to solve the refinement problem for mvPDAs. This program should eventually be integrated into a tool for modal transitions systems called MoTraS [Sto11]. Therefore, to maintain cross-platform compatibility, it is written in Scala and Java, which allows to run on the JVM. The program can also be used on its own from the command line.

Here we will describe the main parts of the program, some finer implementation details, the basic usage and show the performance results of several benchmarks.

3.2 Implementation

The essential parts of the algorithm are be shown here in pseudocode. The main entry point is the function mvPDARefinement shown in figure 3.2. It takes as input two initial processes $P \cdot S$, $Q \cdot T$ and an mvPDA. It initialises the rules with the set of basic rules and enters a loop. As long as there are two rules in the set of rules such that they can be combined into new rules not completely contained in the set of rules, the new rules will be added. This will compute the fixpoint of the set of rules under the combination of rules. Then it returns if the refinement $P \cdot S \leq_m Q \cdot T$ holds by testing if the empty rule is contained from the initial state.

```
1: function mvPDARefinement($P \cdot S, Q \cdot T, mvPDA$)
2:     $initial \leftarrow (P \cdot S, Q \cdot T)$
3:     $rules \leftarrow$ makeRules($mvPDA$)
4:     while $\exists lhsRule, rhsRule \in rules$ : Combine($lhsRule, rhsRule$) $\not\subset rules$ do
5:         $rules \leftarrow rules \cup$ combine($lhsRule, rhsRule$)
6:     end while
7:     return $(initial, \emptyset) \in rules$
8: end function
```

Figure 3.1: Algorithm for testing refinement on an mvPDA

```
1: function MAKERULES($mvPDA = (\Delta_{\text{may}}, \Delta_{\text{must}})$)
2:     $rules \leftarrow \emptyset$
3:     for $P, Q, S, T \in Const(mvPDA), a \in Act(mvPDA)\}$ do
4:         ▷ Attack from left-hand side for may rules
5:         $lhs \leftarrow (P \cdot S, Q \cdot T)$
6:         for $(P \cdot S, a, p') \in \Delta_{\text{may}}$ do
7:             $rhs \leftarrow \emptyset$
8:             for $(Q \cdot T, a, q') \in \Delta_{\text{may}}$ do
9:                 $rhs \leftarrow rhs \cup \{(p', q')\}$
10:            end for
11:            $rules \leftarrow rules \cup \{(lhs, rhs)\}$
12:        end for
13:        ▷ Attack from right-hand side for must rules
14:        $lhs \leftarrow (Q \cdot T, P \cdot S)$
15:        for $(Q \cdot T, a, q') \in \Delta_{\text{must}}$ do
16:            $rhs \leftarrow \emptyset$
17:            for $(P \cdot S, a, p') \in \Delta_{\text{must}}$ do
18:                $rhs \leftarrow rhs \cup \{(p', q')\}$
19:            end for
20:            $rules \leftarrow rules \cup \{(lhs, rhs)\}$
21:        end for
22:    end for
23:    return $rules$
24: end function
```

Figure 3.2: Algorithm for creating the basic attack rules on an mvPDA

```
1: function COMBINE($lhsRule = (lhs, lhsRhsSet), rhsRule = (rhsLhs, rhsSet)$)
2:     $rules \leftarrow \emptyset$
3:     if $\forall rhs \in rhsSet : size(rhs) = 1$ then
4:         for $lhsRhs \in lhsRhsSet : lhsRhs = rhsLhs \cdot p$ do
5:             $newRhs \leftarrow (lhsRhsSet \setminus lhsRhs) \cup \{rhs \cdot p \mid rhs \in rhsSet\}$
6:             $rules \leftarrow rules \cup \{(lhs, newRhs)\}$
7:         end for
8:     end if
9:     return $rules$
10: end function
```

Figure 3.3: Algorithm for combining attack rules

The function MAKERULES shown in figure 3.2 takes an mvPDA as input and rules and returns the attack rules that can be constructed out of the rewrite rules in the mvPDA. This is basically the implementation of the inference rules 1 and 2 for attack rules.

The function COMBINE shown in figure 3.2 then takes two rules and returns all rules that be constructed by combining the two rules. Note that the rules $lhsRule$ and $rhsRule$ are not required to be actually left-hand side and right-hand side rules, but if they are not, simply the empty set is returned. Otherwise all combinations allowed by the inference rules 3 and 4 are formed and returned.

3.3 Soundness and completeness

As the algorithm constructs exactly all the attack rules allowed by the inference rules, soundness follows from theorem 1 and theorem 2. For an input mvPDA with the refinement problem $P \cdot S \leq_m Q \cdot T$, if the algorithm returns **true**, then $P \cdot S \leq_m Q \cdot T$, and if if the algorithm returns **false**, then $\neg(P \cdot S \leq_m Q \cdot T)$.

For completeness we only need to show that the algorithm always terminates. The algorithm never adds a rule twice to its set of rules, and each iteration of the while loop adds at least one rule. The set of possible attack rules over a finite set of constants is finite, and the algorithm only uses constants from the finite mvPDA, therefore it will terminate.

3.4 Complexity

Let $k = |Const|$ be the number of constants appearing in the input mvPDA. For a rule $(p, q) \longrightarrow_a S$, as $(p, q) \in \{(P{\cdot}S, Q{\cdot}T) \mid P, S, Q, T \in Const\}$, there are k^4 possible states for (p, q). F or each $(p', q') \in S$, as $(p', q') \in \{(p, q) \mid |p| \leq 3 \wedge |q| \leq 3 \wedge p, q \text{ sequential}\}$, there are at most k^6 possible states. With the number of subsets S being in $\mathcal{O}(2^{k^6})$, the number of possible rules is in $\mathcal{O}(k^4 2^{k^6})$. An input of size n can define $\mathcal{O}(n)$ constants, therefore the worst-case complexity is $\mathcal{O}(n^4 2^{n^6}) = \mathcal{O}(2^{n^6}) \subseteq$ EXPTIME. The refinement problem on mvPDA is EXPTIME-complete [BK12], indicating there is quite possibly no better solution.

However this is only the worst-case complexity and only needed by mvPDA with rules of a certain structure. For many types of inputs, the algorithm only needs polynomial time.

3.5 Optimizations

The algorithm as given above in pseudocode gives a naive implementation and can be improved in several ways. While these do not reduce the worst-case complexity, on many inputs a significant speedup is measurable. The following are the main optimizations used in the actual implementation.

Worklist algorithm Instead of iterating over the entire set of rules to find matching rules, new rules are added to a worklist. The main loop of the algorithm removes new rules from the worklist one at a time, adds the new rule to the set of rules and combines it with all matching rules. Newly obtained rules are then added to the worklist again.

Hash map lookup Again for finding a matching rules, iterating over all rules can take exponential time. A better approach is to separate the rules into left-hand side rules and right-hand side rules, and for each state $(P \cdot S, Q \cdot T)$, keep a reference to all rules of each type that apply from that state. Specifically, if we have a rule $(p, q) \longrightarrow_a S$, if it is a right-hand side rule keep a reference to that rule from (p, q), and if it is left-hand side rule keep a reference from each $(P{\cdot}S, Q{\cdot}T)$ where $(P{\cdot}S, Q{\cdot}T) \in S$ or $(P{\cdot}S{\cdot}S', Q{\cdot}T{\cdot}T') \in S$. That way, after taking a rule from the worklist, matching can be performed in time linear to the number of matching rules.

Keeping only minimial rules When there are two attack rules $(p, q) \longrightarrow_a S$ and $(p, q) \longrightarrow_a S'$ with $S \subseteq S'$, only the smaller needs rule $(p, q) \longrightarrow_a S$ needs to be kept and $(p, q) \longrightarrow_a S'$ can be removed. If we can obtain $(p, q) \longrightarrow_a \emptyset$ from a sequence that reduces S', we can also obtain it from S. On the other hand, if there is no sequence that reduces S', then there is also no sequence that reduces S. Therefore the correctness of the algorithm is not affected.

Heuristic for combining rules With the optimization to only keep minimal rules, we would like to obtain these as early as possible. While finding the optimal strategy is as hard as solving the problem, a suitable heuristic is to choose rules $(p, q) \longrightarrow_a S$ with the smallest S first. This strategy especially for non-refining process, where we rules have $S = \emptyset$, which always leads to smaller rules. For the implementation, this means using a priority queue as the worklist.

Reachable state exploration Instead of initialising the set of rules with rules from all possible states, we can add rules as we reach a state. Reachability is decidable on PDAs [BEM97] and we can apply that to the combined states of our attack rules. We start with the initial state, and whenever we obtain a new state on the right-hand side as an internal or call state, we add the attack rules from that state. This also reduces the added complexity of combining two mvPDA with no shared states into a single mvPDA.

Early stopping Instead of computing all possible rules and then testing if the empty is obtained, we can stop as soon as we find $(P \cdot S, Q \cdot T) \longrightarrow_a \emptyset$ and return **false**. However, this only improves the runtime if the processes do not refine. If they refine, the algorithm has to explore all the possible rules before returning **true**.

3.6 Usage

The refinement algorithm can be called from the command line via

```
java -jar vmpda-refinement.jar <files>
```

where `<files>` is a list of space-separated input files.

3.6.1 Input

Each ipnut for the program is given in a file containing the processes to test for refinement and the rewrite rules of the mvPDA. The file format used here is similiar to ones used in existing tools [Sic12] to ease the integration with the MoTraS tool. In figure 3.6.1 the grammar is given. For clarity, whitespace is not explicitly noted, but needed between identifiers. At other places it is ignored.

A valid input needs to specify $\langle mprs \rangle$ with the refinement problem and the rules. Note that any mPRS can be defined in the grammer, but the algorithm tests if it is an actual mvPDA and outputs an error otherwise. After parsing the input, the algorithm may rewrite the processes to bring them in a normal form in accordance to the congruence relation of the operators.

mPRS definition

$\langle mprs \rangle$ ::= `mprs` $\langle id \rangle$ `[` $\langle refinement \rangle$ $\langle rule \rangle^*$ `]`

$\langle refinement \rangle$ ::= $\langle process \rangle$ `<=` $\langle process \rangle$

Rule definition

$\langle rule \rangle$ ::= $\langle process \rangle$ $\langle action \rangle$ $\langle ruletype \rangle$ $\langle process \rangle$

$\langle action \rangle$ ::= $\langle id \rangle$

$\langle ruletype \rangle$ :: = $\langle mayrule \rangle$ | $\langle mustrule \rangle$

$\langle mayrule \rangle$ :: = `?`

$\langle mustrule \rangle$ :: = `!`

Process definition

$\langle process \rangle$ ::= $\langle empty \rangle$ | $\langle constant \rangle$ | $\langle parallel \rangle$ | $\langle sequential \rangle$ | `(` $\langle process \rangle$ `)`

$\langle empty \rangle$ ::= `_`

$\langle constant \rangle$ ::= $\langle id \rangle$

$\langle parallel \rangle$ ::= $\langle process \rangle$ `.` $\langle process \rangle$

$\langle sequential \rangle$ ::= $\langle process \rangle$ `|` $\langle process \rangle$

Common definitions

$\langle letter \rangle$ ::= `a` | ... | `z` | `A` | ... | `Z`

$\langle digit \rangle$ ::= `0` | ... | `9`

$\langle id \rangle$ ::= $\langle letter \rangle(\langle letter \rangle \mid \langle digit \rangle)^*$

Figure 3.4: Grammar for the input files

Example 5. *Listing 3.1 gives the input for our vending machine mvPDA from figure 2.1.*

Listing 3.1: Input representing the vending machine mvPDA

```
mprs vpda [
    p.S <= q.S

    p.S coin! p.M.S
    p.M coin! p.M.M
    p.M tea! t
    p.M coffee! c

    t.M tea! t
    t.S coin! p.M.S

    c.M coffee! c
    c.S coin! p.M.S


    q.S coin? q.T.S
    q.S coin? q.C.S
    q.T coin? q.T.T
    q.C coin? q.C.C

    q.T tea! q
    q.T coffee? q

    q.C coffee! q
    q.C tea? q
]
```

3.6.2 Output

After calling the program with a list of input files, the program prints a line with the result of the refinement test for each input. The line consists of a result code, the filename and the running time in case of successful execution. The result codes are explained in figure 3.6.2.

Possible causes for exceptions are I/O Exceptions when reading the file, parsing exceptions for malformed input or illegal argument exceptions when the input does not represent an mvPDA.

Result code	Meaning
0	The processes refine
1	The processes do not refine
E	There was an exception

Figure 3.5: Result codes

Example 6. *Listing 3.2 shows the output from the run of the program on three input files. The first one is the vending machine mvPDA and the algorithm shows it does not refine, the second one is refining example and the third is an invalid input, where the given mPRS is not an mvVPDA.*

Listing 3.2: Usage example

```
$ java -jar vmpda-refinement.jar vendingmachine.mprs
    vpda.mprs non_vpda.mprs
[0] src/main/resources/vendingmachine.mprs (0.293342559 s)
[1] src/main/resources/vpda.mprs (7.682377187 s)
[E] src/main/resources/non_vpda.mprs
    (java.lang.IllegalArgumentException: Given mPRS is not
    an mvPDA)
```

3.7 Performance evaluation

4 Conclusion

4.1 Main results

4.2 Further extensions

Bibliography

[AEM04] Rajeev Alur, Kousha Etessami, and P. Madhusudan, *A temporal logic of nested calls and returns*, TACAS (Kurt Jensen and Andreas Podelski, eds.), Lecture Notes in Computer Science, vol. 2988, Springer, 2004, pp. 467--481.

[AM04] Rajeev Alur and P. Madhusudan, *Visibly pushdown languages*, STOC (New York), ACM Press, June 13--15 2004, pp. 202--211.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler, *Reachability analysis of pushdown automata: Application to model-checking*, CONCUR, 1997, pp. 135--150.

[BK12] Nikola Benes and Jan Kretínský, *Modal process rewrite systems*, ICTAC (Abhik Roychoudhury and Meenakshi D'Souza, eds.), Lecture Notes in Computer Science, vol. 7521, Springer, 2012, pp. 120--135.

[Esp01] Javier Esparza, *Grammars as processes*, Lecture Notes in Computer Science **2300** (2001), 277--298.

[GH94] Jan Friso Groote and Hans Hüttel, *Undecidable equivalences for basic process algebra*, Inf. Comput. **115** (1994), no. 2, 354--371.

[KMV07] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan, *Visibly pushdown automata for streaming XML*, WWW (Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, eds.), ACM, 2007, pp. 1053--1062.

[LT88] Kim Guldstrand Larsen and Bent Thomsen, *A modal process logic*, LICS, 1988, pp. 203--210.

[May00] Richard Mayr, *Process rewrite systems*, Inf. Comput **156** (2000), no. 1-2, 264--286.

[Sic12] Salomon Sickert, *Refinement algorithms for parametric modal transition systems*, Bachelor's thesis, Technische Universität München, 2012.

[Srb06] Jirí Srba, *Visibly pushdown automata: From language equivalence to simulation and bisimulation*, CSL (Zoltán Ésik, ed.), Lecture Notes in Computer Science, vol. 4207, Springer, 2006, pp. 89--103.

[Sto11] Martin Stoll, *Motras : A tool for modal transition systems*, Bachelor's thesis, Technische Universität München, 2011.