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
Date:	April 15th, 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

1 Introduction

As an extension to labelled transition systems, modal transition systems (MTS) [LT88] have been widely used, especially in model checking. They provide a way to describe system specifications in a way that allows stepwise refinement and composition of several refinements. An MTS has two types of transitions, *may* transitions, which are admissible, and *must* transitions, which are necessary. A refinement of an MTS should then 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. A central problem then is to decide whether modal refinement holds between two systems.

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 automata (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 basic process algebras (BPA), already simulation is undecidable [GH94]. As refinement is a generalization of both simulation and bisimulation, it is also undecidable on mBPA and mPDA.

However there is the subclass of visibly pushdown automata (vPDA), 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 vPDA as well to allow modal specifcations and abstractions. This requires a way to test refinement between these. Fortunately, not only simulation and bisimulation is decidable [Srb06], but also modal refinement [BK12]. Based on that result, we present an algorithm with its underlying theory to decides modal refinement on mvPDA.

2 Theory

2.1 Processes

As a unified formalism to describe many classes of infinite state systems, process rewrite systems were introduced by [May00]. They specify rewrite rules on processes, which states of the systems. The syntax here on processes is taken from [Esp01].

Definition 1 (Process)**.** The set of *processes* $\mathcal{P}$ over a set of constants $Const = \{X, Y, ...\}$ 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 $\|$.

The class of proceses just obtained with rule 0,1 and S are called sequential processes, while processes just obtained with rule 0,1 and P are called parallel processes.

From here on lowercase letters $p, q, ...$ will denote any process, while uppercase letters $P, Q, ...$ will denote single process constants.

Example 1. *The state of a pushdown automaton, given by a control state P and a stack $A_1 A_2 ... A_n$, where A_1 is the top of the stack, can be given by a sequential process $P \cdot A_1 \cdot A_2 \cdot ... \cdot A_n$. As an intuition, a transition from P with the symbol a to Q, by replacing A_1 on the stack with $B_1 B_2$, could be given by a rewrite rule $P \cdot A_1 \xrightarrow{a} Q \cdot B_1 \cdot B_2$. Applying this rule on the state yields $Q \cdot B_1 \cdot B_2 \cdot A_2 \cdot ... \cdot A_n$. The formal definition for rewrite rules is given later.*

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 equal to the number of constants appearing in the process.

2.2 Modal transition system

Modal transitions systems and subsequently modal refinement are defined here as in [BK12].

Definition 3 (Modal transition system)**.** A *modal transition system (MTS)* over an action alphabet $Act = \{a, b, ...\}$ 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 $p \xrightarrow{\text{a}} q$.

When using the term of a process in the context of an MTS, we will have its meaning include the underlying transition system, That way we can talk about refinement of a process by another process, without mentioning their originating systems.

2.3 Modal refinement

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 as $p \leq_m q$, 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. $(p', q') \in \mathcal{R}$.

Out of simplicity, we only regard refinement 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.

The intuition for a process p refining another process q is that p is a more specific than q, while q on the contrary is more abstract than p. Modal refinement can then be used to compare different specifications of a system or to obtain an *implementation*, that is an MTS with $\dashrightarrow = \longrightarrow$.

Modal refinement can also be seen as a refinement game, similiar to the standard simulation or bisimulation game [Srb06]. The states of the game are pairs of processes (p, q), and in each round a process from one side plays an attacking transition, to which the other process answers with a defending transition. This leads to a new state, from which the game continues. The attacker wins if at some state the defender can not play any more transitions, otherwise the defender wins.

An *attacking transition* from the state (p, q) is any transition of the form $p \overset{\mathrm{a}}{\dashrightarrow} p'$ or $q \xrightarrow{\mathrm{a}} q'$. Given that attacking transition, any transition of the form $q \overset{\mathrm{a}}{\dashrightarrow} q'$ or $p \xrightarrow{\mathrm{a}} p'$ matching the action and the transition type is a *defending transition.* That choice of transitions then leads to the state (p', q').

Then $p \leq_m q$ holds if and only if there is no winning strategy for the attacker from (p, q), i.e. a sequence of attacking transitions such that for every choice of defending transition, a state (p', q') is reached from which there is an attacking transition but no defending transition. The complete characterization for this is given in [BK12].

2.4 Modal process rewrite system

Modal transitions systems are defined here as a straightforward extension of process rewrite systems [May00, Esp01], with their modal subclasses defined analogously.

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:

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

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

2.5 Visibly pushdown automaton

Visibly pushdown automata are a subclass of PDA that partition their action alphabet into call actions, internal actions and return actions. Here they will be represented by a PRS in the same form used to represent PDA.

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 with the same action alphabet partition.

The rules of a vPDA are called return rules if they pop a symbol from the stack, internal rules if they only change the top symbol or call rules if they push a symbol to the stack. A vPDA always makes the type of rule used visible through the action. In contrast to a PDA, this means that two vPDA performing the same actions will always have the same number of symbols on the stack. This will be crucial to decide refinement between them.

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

Example 2. *Suppose we have a modular specification for a vending machine that can offer either tea, coffee, or both. They should only be offered after inserting coins, although that is optional. However after inserting a number of coins, the machine either has to offer one tea for each coin inserted or one coffee for each coin inserted. It may also offer, but does not need to. Then we obtain a proposed implementation and want to test if it refines the specification.*

Figure 2.1 shows both implementation and specification modeled by an mvPDA. Note that may transitions are implied by the must transitions.

The process $Q \cdot S$ on the right is the specification. After inserting the first coin, the machine non-deterministically chooses either tea or coffee by pushing T or C on the stack. This symbol is added for every additional coin. It can be popped for the chosen beverage and may also be popped for the other one.

The process $P \cdot S$ on the left is the implementation. It avoids non-determinism, so after inserting coins it just stores that information on the stack with the symbol M. When it offers a beverage, it locks in the chosen type by either going to the state T or C. From those states it can only offer the chosen beverage, until no more coins are left.

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

Figure 2.1: Vending machine mvPDA

Example 3. *We can look at some refinement problems on the mvPDA from figure 2.1. First we want to decide if $T \cdot M \leq_m Q \cdot T$ 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 there are no more transitions from the resulting state (T, Q), therefore $T \cdot M \leq_m Q \cdot T$ holds.*

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, the strategies for the attacker can be seen as trees. That will be done through attack trees, as a representation of partially or fully explored strategies. These 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, child states and child trees. 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 child 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 label of the root, $T_O = O$ for the set of child states and $T_C = C$ for the set of child trees.

The set of attack trees $\mathcal{T}$ constructable from an MTS $(\mathcal{P}, \dashrightarrow, \longrightarrow)$ is defined 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 construct an initial tree for an attacking transition with edges to states for each possible defending transitions, while rule 3 replaces an open edge to a state with a tree having that state as its root.

As we can see from the construction rules, every node in a tree has a corresponding attacking transition, while for each edge from that node there is an applicable defending transition. Similarly, as the total number of edges never changes, there is an edge for each applicable defending transition. Therefore nodes can be identified with attacking transitions and edges with defending transitions.

For an attack tree T, the set of all *subtrees* of T, including T itself, is given 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 $open(T) = T_O \cup \left(\bigcup_{T' \in T_C} open(T')\right)$.

A tree is said to be *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 any open states in the tree can be replaced by a matching 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 $open(S) = open(T) \setminus \{R_r\} \cup open(R)$.*

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

1. $n = 0$: Then $R_r \in T_O$ and $R_r \notin \mathit{open}(T')$ 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 $\mathit{open}(S) = \mathit{open}(T) \setminus \{R_r\} \cup \mathit{open}(R)$.

2. $n \geq 1$: Then there is $T' \in T_C$ such that $R_r \in \mathit{open}(T')$. T' 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 $\mathit{open}(S') = \mathit{open}(T') \setminus \{R_r\} \cup \mathit{open}(R)$

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

   If $R_r \neq \mathit{open}(T'')$, then $\mathit{open}(T'') = \mathit{open}(T) \setminus \{R_r\} \cup \mathit{open}(R)$ 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 $\mathit{open}(S) = \mathit{open}(T'') \setminus \{R_r\} \cup \mathit{open}(R) = \mathit{open}(T) \setminus \{R_r\} \cup \mathit{open}(R)$.

□

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 $p, q \in \mathcal{P}$:*

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

Proof. $\Longrightarrow$: 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 transitions. 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 $n = |\mathit{subtrees}(T)|$:

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 $\mathcal{R} := \{(p', q') \mid \neg\exists T : T_r = (p', q') \wedge \mathit{closed}(T)\}$ 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 $\mathit{open}(T)$ is minimal with regard to the inclusion order. There exists $(p', q') \in \mathit{open}(T)$ 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 $\mathit{open}(T'') = \mathit{open}(T) \setminus \{(p', q')\} \subsetneq \mathit{open}(T)$ in contradiction to the minimality of $\mathit{open}(T)$. So for the attacking transition from (p, q), there is a defending transition to (p', q') with $(p', q') \in \mathcal{R}$. □

Example 4. *For the MTS induced by the vending machine mvPDA from figure 2.1, attack trees for certain states are displayed in figure 2.2. These are obtained by the basic rules 1 and 3. 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.*

With rule 3, we can combine all these trees to construct 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 directly from the tree by the associated attacking and defending transitions, until a leaf node is reached which has no more defending transitions.

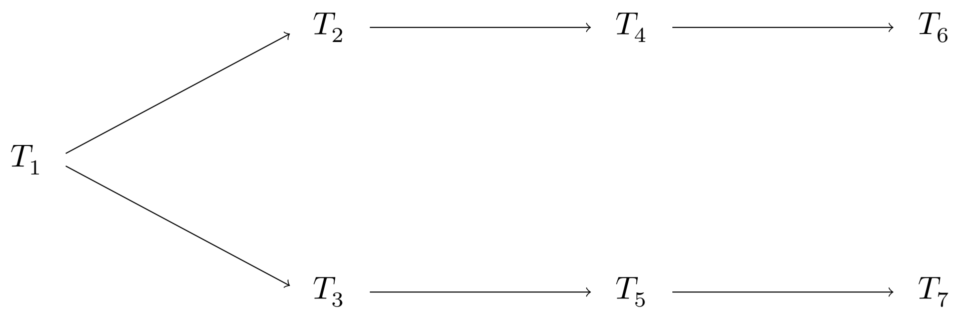

Figure 2.3: Combined attack tree for the vending machine MTS

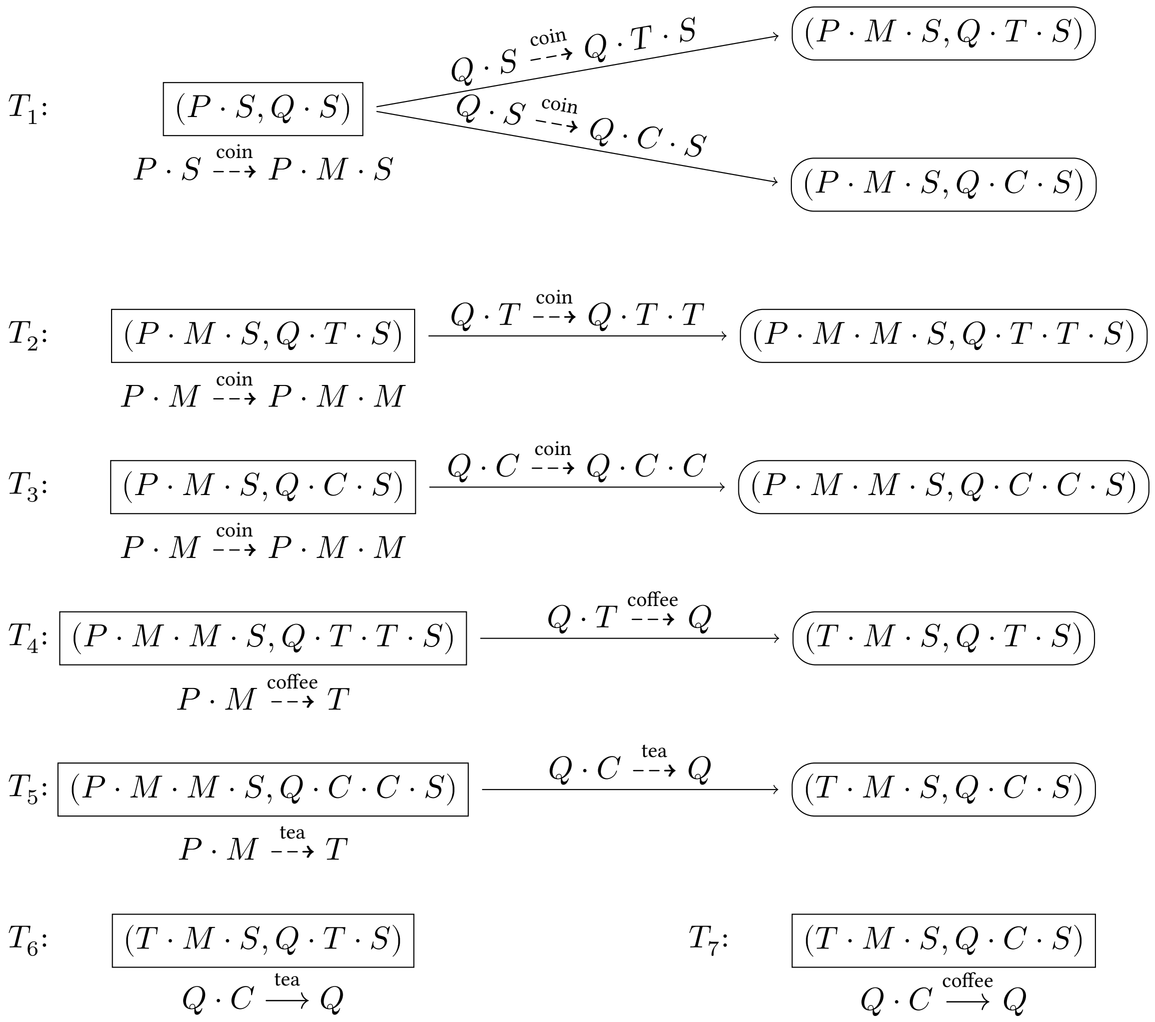
T_1:
$(P \cdot S, Q \cdot S)$
$P \cdot S \overset{\text{coin}}{\dashrightarrow} P \cdot M \cdot S$
$Q \cdot S \overset{\text{coin}}{\dashrightarrow} Q \cdot T \cdot S$
$(P \cdot M \cdot S, Q \cdot T \cdot S)$
$Q \cdot S \overset{\text{coin}}{\dashrightarrow} Q \cdot C \cdot S$
$(P \cdot M \cdot S, Q \cdot C \cdot S)$
T_2:
$(P \cdot M \cdot S, Q \cdot T \cdot S)$
$P \cdot M \overset{\text{coin}}{\dashrightarrow} P \cdot M \cdot M$
$Q \cdot T \overset{\text{coin}}{\dashrightarrow} Q \cdot T \cdot T$
$(P \cdot M \cdot M \cdot S, Q \cdot T \cdot T \cdot S)$
T_3:
$(P \cdot M \cdot S, Q \cdot C \cdot S)$
$P \cdot M \overset{\text{coin}}{\dashrightarrow} P \cdot M \cdot M$
$Q \cdot C \overset{\text{coin}}{\dashrightarrow} Q \cdot C \cdot C$
$(P \cdot M \cdot M \cdot S, Q \cdot C \cdot C \cdot S)$
T_4:
$(P \cdot M \cdot M \cdot S, Q \cdot T \cdot T \cdot S)$
$P \cdot M \overset{\text{coffee}}{\dashrightarrow} T$
$Q \cdot T \overset{\text{coffee}}{\dashrightarrow} Q$
$(T \cdot M \cdot S, Q \cdot T \cdot S)$
T_5:
$(P \cdot M \cdot M \cdot S, Q \cdot C \cdot C \cdot S)$
$P \cdot M \overset{\text{tea}}{\dashrightarrow} T$
$Q \cdot C \overset{\text{tea}}{\dashrightarrow} Q$
$(T \cdot M \cdot S, Q \cdot C \cdot S)$
T_6:
$(T \cdot M \cdot S, Q \cdot T \cdot S)$
$Q \cdot C \xrightarrow{\text{tea}} Q$
T_7:
$(T \cdot M \cdot S, Q \cdot C \cdot S)$
$Q \cdot C \xrightarrow{\text{coffee}} Q$

Figure 2.2: Basic attack trees for the vending machine MTS

2.7 Attack rule

Attack trees can represent strategies for any MTS. However, their states can be arbitrarily large and there is an infinite number of trees for infinite state systems. Attack rules are a similiar concept with a finite state set specifically for mvPDA and can be used to represent certain 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 $(p, q) \longrightarrow_a S$.

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

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

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

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

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

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 $(p,q) \longrightarrow_a S \uplus \{(p',q')\}$ on the left and a rule $(p',q') \longrightarrow_a S'$ on the right, it always holds that $|p'| = 2$ or $|p'|$ = 3 and for all $(p'',q'') \in S'$ $|p''| = 1$. A rule $p \longrightarrow_a S$ is then 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. This, combined with the inductive nature of the rules, will be used to develop an algorithm to decide refinement later.

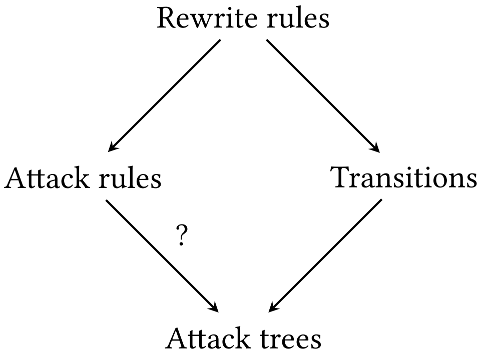

Figure 2.4: Relation between concepts on mvPDA and MTS

Our goal is now to find a relation between attack rules and attack trees. Figure 2.4 shows the relation between different concepts of PRS and MTS. For an mvPDA with a set of rewrite rules, an MTS is induced with a set of transitions, from which attack trees are induced. Also from the rewrite rules, the attack rules are induced. The missing connection is from attack rules to attack trees, which we need to close. For that, first we need show that the relation between transitions and lifted versions of the transitions is linear for an mvPDA.

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

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

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 $r \cdot s'' \overset{\text{a}}{\dashrightarrow} r' \cdot s'' = p \overset{\text{a}}{\dashrightarrow} p'$. The same holds for $q \cdot t \xrightarrow{\text{a}} q' \cdot t$. □

This can be applied to attack trees to lift their states with added processes.

Lemma 3. *Given an MTS induced by an 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. In each base case in the construction of T, the attacking transition can be lifted with s or t. For each defending transition, there is exactly one lifted transition by lemma 2. Each other construction rule can still be applied on the lifted trees and we obtain the tree R. □

While 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 $\mathbb{P}$ of an attack tree T is a set of subtrees $\mathbb{P} \subseteq subtree(T)$ with $T \in \mathbb{P}$.

For $R_1, R_2 \in \mathbb{P}$, there is a partial ordering $R_1 \leq R_2 \iff R_1 \in subtree(R_2)$ and consequently $R_1 < R_2 \iff R_1 \leq R_2 \land R_1 \neq R_2$. Under that relation, the partition successors of $R \in P$ given $\mathbb{P}$ are $succ_{\mathbb{P}}(R) = \{R' \in \mathbb{P} \mid R' < R \land \neg\exists R'' : R' < R'' \land R'' < R\}$.

An attack rule should then correspond to a part, or represent it, if it can be lifted such that it leads from the root node of the part of the part to all of its successors.

Definition 10 (Part represented by an attack rule). A subtree $R \in \mathbb{P}$ 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 $\{R'_r \mid R' \in succ_{\mathbb{P}}(R)\} = \{(p' \cdot s, q' \cdot t) \mid (p', q') \in S\}$

Now we can prove our main theorem, allowing us to decide existance of a closed tree by using attack rules.

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

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

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 $\mathbb{P} = \{T'_1, ..., T'_n\}$ of T with n elements 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 $|p| = |q| = 2$, As $(p, q) = (P \cdot T, Q \cdot R)$ and $succ_{\mathbb{P}}(T) = \emptyset$ it follows that $S = \emptyset$. Then the rule is $(P \cdot T, Q \cdot R) \longrightarrow_a \emptyset$.

2. $n > 1$: For T as the root part in $\mathbb{P}$, there is $T' \in succ_{\mathbb{P}}(T)$ as $n > 1$ Let $T'_r = (p', q')$ and $a = (P \cdot S, Q \cdot R) \longrightarrow_a S$ be the rule representing T. We have $(p', q') \in S$ and necessarily $|p'| = |q'| \geq 2$, as otherwise there would be no transition applicable from that state and therefore T'' would not exist. So a is a left-hand side rule.

   For every part $T'' \in \mathbb{P}$ with $succ_{\mathbb{P}}(T'') = \emptyset$, the representing rule $b = (p, q) \to S$ has $S = \emptyset$, so it is a right-hand side rule. Every path in $\mathbb{P}$ eventually leads to such a part.

   Then by following the successors of the parts from T, eventually there will be a part T' succeded by a part T'' such that the rule representing T' is a left-hand side rule and the rule representing T'' is a right-hand side rule.

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

   Let $(p, q) \longrightarrow_a S$ be the rule representing T' and $(p', q') \longrightarrow_a S'$ be the rule representing T''. By representation, there are $s, t \in \mathcal{P}$, $(p'', q'') \in S$ with $T''_r = (p'' \cdot s, q'' \cdot t)$ and $s', t' \in \mathcal{P}$ 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 $\{(p' \cdot s, q' \cdot t) \mid (p', q') \in S \setminus \{(p'', q'')\} \cup S'\} = \{T'_r \mid succ_{\mathbb{P}'}(T')\}$, it represents T' in $\mathbb{P}'$.

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 $\{(p' \cdot s, q' \cdot t) \mid (p', q') \in S \setminus \{(p'', q'')\}\} \cup \{(p'' \cdot P' \cdot s, q'' \cdot Q' \cdot t) \mid (p'', q'') \in S'\} = \{T'_r \mid succ_{\mathbb{P}'}(T')\}$, it represents T' in $\mathbb{P}'$.

Then as $\mathbb{P}'$ 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. Take $\mathbb{P} = subtrees(T)$. For each $T' \in \mathbb{P}$, there is an attacking transition from T'_r which induced T'. As $succ_P(T') = T'_C$, 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 $|p| = |q| = 2$ for some $s, t \in \mathcal{P}$. By lemma 2, for each transition $p \cdot s \overset{\text{a}}{\dashrightarrow} p' \cdot s$ there is an inducing $(p, a, p') \in \Delta_{\text{may}}$ and for each $q \cdot t \overset{\text{a}}{\dashrightarrow} q' \cdot t$ there is an inducing $(q, a, q') \in \Delta_{\text{may}}$. The same holds for $\overset{\text{a}}{\longrightarrow}$ 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 $p \overset{\text{a}}{\dashrightarrow} p'$ and for every $(q, a, q') \in \Delta_{\text{may}}$ there is an induced defending transition $q \overset{\text{a}}{\dashrightarrow} q'$. Then $S = \{(p', q') | q \overset{\text{a}}{\dashrightarrow} q'\}$ 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 $q \overset{\text{a}}{\longrightarrow} q'$ and for every $(p, a, p') \in \Delta_{\text{must}}$ there is an induced defending transition $p \overset{\text{a}}{\longrightarrow} p'$. Then $S = \{(p', q') | p \overset{\text{a}}{\longrightarrow} p'\}$ 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 $open(T) = \emptyset$. □

Example 5. *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.*

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

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

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

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

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

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

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

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

Initially we take the first partition of the attack tree, where every part has a basic attack rule representing it, as shown in figure 2.5. 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 rules we can combine are the ones from T_2 with T_4 and T_3 with T_5. Then we obtain the second partition and rules shown in figure 2.6.

After that we can combine the rules from T_2 with T_6 and from T_3 with T_7. This results in the third partition and rules shown in figure 2.7. Finally we combine T_1 first with T_2 and then with T_3 to obtain the fourth partition with the rule shown in figure 2.8. This partition representes the whole tree, and as $(P \cdot S, Q \cdot S) \longrightarrow_a \emptyset$, we can decide that there is a winning strategy from $(P \cdot S, Q \cdot S)$, therefore the states do not refine.

$$\begin{array}{ll}
T_1 & \boxed{(P \cdot S, Q \cdot S) \longrightarrow_a \{(P \cdot M \cdot S, Q \cdot C \cdot S), (P \cdot M \cdot S, Q \cdot T \cdot S)\}} \\
T_2 \longrightarrow T_4 & \boxed{(P \cdot M, Q \cdot T) \longrightarrow_a \{(C \cdot M, Q \cdot T)\}} \\
T_3 \longrightarrow T_5 & \boxed{(P \cdot M, Q \cdot C) \longrightarrow_a \{(T \cdot M, Q \cdot C)\}} \\
T_6 & \boxed{(C \cdot M, Q \cdot T) \longrightarrow_a \emptyset} \\
T_7 & \boxed{(T \cdot M, Q \cdot C) \longrightarrow_a \emptyset}
\end{array}$$

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

$$\begin{array}{ll}
T_1 & \boxed{(P \cdot S, Q \cdot S) \longrightarrow_a \{(P \cdot M \cdot S, Q \cdot C \cdot S), (P \cdot M \cdot S, Q \cdot T \cdot S)\}} \\
T_2 \longrightarrow T_4 \longrightarrow T_6 & \boxed{(P \cdot M, Q \cdot T) \longrightarrow_a \emptyset\}} \\
T_3 \longrightarrow T_5 \longrightarrow T_7 & \boxed{(P \cdot M, Q \cdot C) \longrightarrow_a \emptyset}
\end{array}$$

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

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

Figure 2.8: Attack rules for fourth partition of vending machine tree

3 The refinement algorithm

The given theory then lead to the development of a program implementing an algorithm for solving the refinement problem on mvPDA. This program should eventually be integrated into the MoTraS tool for modal transitions systems [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 as a standalone tool from the command line.

The basic idea of the algorithm is computing the set of all attack rules for a given mvPDA, by which it can decide refinement. Here the parts of the program are described as well as some finer implementation details, the basic usage and the performance results of several benchmarks is shown.

3.1 Implementation

The main entry point is the function mvPDARefinement shown in pseudocode in figure 3.1. It takes as input two processes $P \cdot S$, $Q \cdot T$ and an mvPDA. Then i 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, which are not completely contained in the set of rules, the new rules will be added. This will compute the fixpoint of the 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 from the initial state is contained in the rules.

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

Figure 3.1: Algorithm for testing refinement on an mvPDA

```
function makeRules($mvPDA = (\Delta_{\text{may}}, \Delta_{\text{must}})$)
    $rules \leftarrow \emptyset$
    for $P, Q, S, T \in Const(mvPDA), a \in Act(mvPDA)\}$ do
        ▷ Attack from left-hand side for may rules
        $lhs \leftarrow (P \cdot S, Q \cdot T)$
        for $(P \cdot S, a, p') \in \Delta_{\text{may}}$ do
            $rhs \leftarrow \emptyset$
            for $(Q \cdot T, a, q') \in \Delta_{\text{may}}$ do
                $rhs \leftarrow rhs \cup \{(p', q')\}$
            end for
            $rules \leftarrow rules \cup \{(lhs, rhs)\}$
        end for
        ▷ Attack from right-hand side for must rules
        $lhs \leftarrow (Q \cdot T, P \cdot S)$
        for $(Q \cdot T, a, q') \in \Delta_{\text{must}}$ do
            $rhs \leftarrow \emptyset$
            for $(P \cdot S, a, p') \in \Delta_{\text{must}}$ do
                $rhs \leftarrow rhs \cup \{(p', q')\}$
            end for
            $rules \leftarrow rules \cup \{(lhs, rhs)\}$
        end for
    end for
    return $rules$
end function
```

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

```
function combine($lhsRule = (lhs, lhsRhsSet), rhsRule = (rhsLhs, rhsSet)$)
    $rules \leftarrow \emptyset$
    if $\forall rhs \in rhsSet : size(rhs) = 1$ then
        for $lhsRhs \in lhsRhsSet : lhsRhs = rhsLhs \cdot p$ do
            $newRhs \leftarrow (lhsRhsSet \setminus lhsRhs) \cup \{rhs \cdot p \mid rhs \in rhsSet\}$
            $rules \leftarrow rules \cup \{(lhs, newRhs)\}$
        end for
    end if
    return $rules$
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. For each state $(P \cdot S, Q \cdot T)$ and each possible attack rewrite rule, the right-hand side is created by collecting the new states for all defending rewrite rules.

The function COMBINE shown in figure 3.2 then takes two rules and returns all rules that can 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.

3.2 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 $P \cdot S \overset{?}{\leq}_m Q \cdot T$, if the algorithm returns **true**, then $P \cdot S \leq_m Q \cdot T$ holds, and if if the algorithm returns **false**, then $(P \cdot S \leq_m Q \cdot T)$ does not hold.

For completeness only termination needs to be shown. 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.

3.3 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 (p, q). For 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 $\mathcal{O}\left(2^{k^6}\right)$, the number of possible rules is in $\mathcal{O}\left(k^4 2^{k^6}\right)$.

The main loop of the algorithm is executed at most once for every possible rules. Its body can iterate over every rule created so far a constant amount of times, each time taking time polynomial with regard to the size of a single rule. An input of size n can define $k \in \mathcal{O}(n)$ constants, therefore the worst-case complexity is $\mathcal{O}\left(\left(n^4 2^{n^6}\right)^c\right) = \mathcal{O}\left(n^{4c} 2^{c \cdot n^6}\right) = 2^{\mathcal{O}(n^6)} \subseteq$ EXPTIME for some constant $c \geq 1$. The refinement problem on mvPDA is also EXPTIME-complete, as shown in [BK12].

However the exponent could still be reduced. A similiar problem in [Wal96] suggest $2^{\mathcal{O}(n^2}$, and testing implied that actually $O\left(k^3\right)$ rules are necessary to construct k^6 different right-hand sides from a state. Still this remains to be shown. Further for many types inputs where the global branching degree is constant only polynomial time is needed, as shown later.

3.4 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 Also finding a matching rule, 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, for 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. However there still might be an exponential 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 rule $(p, q) \longrightarrow_a S$ needs to be kept and $(p, q) \longrightarrow_a S'$ can be removed. If $(p, q) \longrightarrow_a \emptyset$ can be obtained from a sequence that reduces S', it can also be obtained 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, these should be obtained as early as possible. While finding the optimal strategy is as hard, a suitable heuristic is to choose rules $(p, q) \longrightarrow_a S$ with the smallest S first. This strategy works especially for non-refining process, where there are rules with $S = \emptyset$, 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, only rules reachable from the initial state are added. Reachability is decidable on PDA [BEM97] and that can be applied to the combined states of the attack rules. The initialization starts with rules from the initial state and whenever a new state is obtained on the right-hand side as an internal or call state, the basic attack rules from that state are added. 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, the algorithm can stop and return **false** as soon as it finds $(P \cdot S, Q \cdot T) \longrightarrow_a \emptyset$. 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**.

3.5 Usage

The refinement algorithm can be called from the command line via

```
java -jar mvpda-refinement.jar [-v] [FILE]...
```

where `[FILE]...` is a list of space-separated input files. The optional switch `-v` turns on verbose mode, where each generated attack rule is printed to the output. This can be used for informative or debugging purposes. Otherwise simply the result of the refinement is shown.

3.5.1 Input

Each input 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 the format used in existing tools [Sic12] for MoTraS. In figure 3.4 the grammar is given. For clarity, whitespace is not explicitly noted, but needed between identifiers. At other places it is ignored.

mPRS definition

⟨*mprs*⟩ ::= `mprs` ⟨*id*⟩ `[` ⟨*refinement*⟩ ⟨*rule*⟩* `]`

⟨*refinement*⟩ ::= ⟨*process*⟩ `<=` ⟨*process*⟩

Rule definition

⟨*rule*⟩ ::= ⟨*process*⟩ ⟨*action*⟩ ⟨*ruletype*⟩ ⟨*process*⟩

⟨*action*⟩ ::= ⟨*id*⟩

⟨*ruletype*⟩ :: = ⟨*mayrule*⟩ | ⟨*mustrule*⟩

⟨*mayrule*⟩ :: = `?`

⟨*mustrule*⟩ :: = `!`

Process definition

⟨*process*⟩ ::= ⟨*empty*⟩ | ⟨*constant*⟩ | ⟨*parallel*⟩ | ⟨*sequential*⟩ | `(` ⟨*process*⟩ `)`

⟨*empty*⟩ ::= `_`

⟨*constant*⟩ ::= ⟨*id*⟩

⟨*parallel*⟩ ::= ⟨*process*⟩ `.` ⟨*process*⟩

⟨*sequential*⟩ ::= ⟨*process*⟩ `|` ⟨*process*⟩

Common definitions

⟨*letter*⟩ ::= `a` | ... | `z` | `A` | ... | `Z`

⟨*digit*⟩ ::= `0` | ... | `9`

⟨*id*⟩ ::= ⟨*letter*⟩(⟨*letter*⟩ | ⟨*digit*⟩)*

Figure 3.4: Grammar for the input files

A valid input needs to contain $\langle mprs \rangle$ with the refinement problem and the rules. Note that any mPRS can be defined by 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.

Example 6. *Listing 3.5 gives the input for the vending machine mvPDA from figure 2.1.*

```
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
]
```

Figure 3.5: Input representing the vending machine mvPDA

3.5.2 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.

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

Figure 3.6: Result codes

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

Example 7. *Figure 3.7 shows the output from the run of the program on three input files. The first one is the vending machine mvPDA with the non-refining processes, the second one is a refining example and the third one is an invalid input.*

```
$ java -jar mvpda-refinement.jar vending_machine.mprs
    vpda.mprs non_vpda.mprs
[0] vending_machine.mprs (0.097 s)
[1] vpda.mprs (12.637 s)
[E] non_vpda.mprs (java.lang.IllegalArgumentException:
   Given mPRS is not an mvPDA: action alphabet can not be
   partitioned)
```

Figure 3.7: Usage example

3.6 Performance evaluation

As seen in the complexity analysis, the runtime of the algorithm could possibly be exponential, which is quickly intractable even on small data. Here different families of input data will be tested to obtain some conditions that can cause exponential runtime.

The construction rules for attack rules show that the main cause for exponential runtime are attack rules with a large right-hand side S. If a rule is applicable for every element of S, it can lead to $2^{|S|}$ new rules. The size of S is determined by the number of rules applicable from a state, which is the branching degree from a state. Therefore mvPDA with varying degrees of branching and non-determinism for both the attacking and defending process will be analyzed.

In each of the following benchmarks, a family of mvPDA parametrized by an $n \in \mathbb{N}$ and a boolean ref is given. On them the refinement problem $P_0 \cdot S \overset{?}{\leq_m} Q_0 \cdot S$ was tested. By construction, they have a number of rules linear in n and the refinement problem holds if ref is true.

To force construction of attack rules with $|S|$ of certain sizes, it is enough to consider mvPDA with the action alphabet $Act = \{r, i, c\}$, for return, internal and call rules, respectively, and only may rules, as both action and rule type are discarded when constructing attack rules. The attacking process will then always be $P_0 \cdot S$ and the defending process $Q_0 \cdot S$.

High local branching degree For the first family of mvPDA, a process is constructed that has a high branching degree from a single state. With the mvPDA defined in figure 3.8, from $Q_0 \cdot S$ there are n defending transitions applicable to the rule attacking transition $P_0 \cdot S \overset{\mathrm{c}}{\dashrightarrow} P_1 \cdot S \cdot S$. This will lead to an attack rule $(P_0 \cdot S, Q_0 \cdot S) \longrightarrow_a S'$ with $|S'| = n$. For each $(P_0 \cdot S \cdot S, Q_0 \cdot S \cdot S_i) \in S'$, there is a right-hand side applicable and therefore an exponential number of possible rules.

$$
\begin{array}{lll@{\qquad}ll}
P_0 \cdot S \overset{\mathrm{c}}{\dashrightarrow} P_1 \cdot S \cdot S & & & Q_0 \cdot S \overset{\mathrm{c}}{\dashrightarrow} Q_1 \cdot S \cdot S_i & \forall 0 \leq i < n \\
P_0 \cdot S \overset{\mathrm{r}}{\dashrightarrow} P_2 & & & Q_1 \cdot S \overset{\mathrm{r}}{\dashrightarrow} Q_2 & \\
P_2 \cdot S \overset{\mathrm{c}}{\dashrightarrow} P_3 \cdot S \cdot S & & & Q_2 \cdot S_i \overset{\mathrm{c}}{\dashrightarrow} Q_3 \cdot S \cdot S_i & \forall 0 \leq i < n \\
P_3 \cdot S \overset{\mathrm{i}}{\dashrightarrow} P_4 \cdot S & & & Q_3 \cdot S \overset{\mathrm{i}}{\dashrightarrow} Q_4 \cdot S_i & \forall 0 \leq i \leq n \\
P_4 \cdot S \overset{\mathrm{r}}{\dashrightarrow} P_5 & \text{if } \neg ref & & &
\end{array}
$$

Figure 3.8: mvPDA with high local branching

Figure 3.9 shows the runtime and rules benchmark on this mvPDA on a logarithmic scale. As expected, both grow exponentially with regard to n.

Note that in the non-refining case, there is an attack tree with a small depth, which could be reduced in linear time by a certain combination of attack rules. However, as the construction creates another rule with degree $n + 1$ from each $(P_2 \cdot S, Q.S_i)$, with the combination heuristic this rule will not be used until all the combinations of the first rule are created. If another heuristic were to be used, the runtime in the non-refining case could be linear.

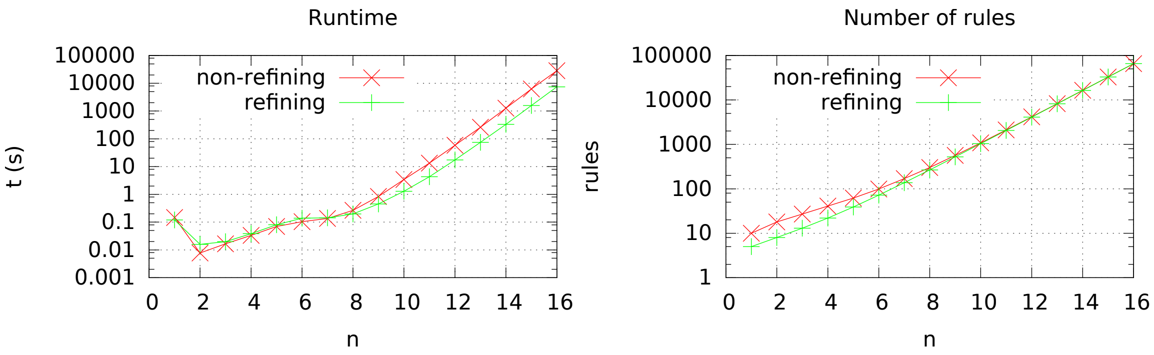

Figure 3.9: Benchmark for mvPDA with high local branching

Constant local branching For this family, the branching degree from any single state is restricted. Figure 3.10 defines an mvPDA where there are at most two rules applicable from any state. The process from $P_0 \cdot S$ can perform n call transitions, one internal transition and n return transitions. These can all be matched by a rule from $Q_0 \cdot S$, with each $Q_i \cdot S$ reachable after n transitions for for $0 \leq i \leq n$. This again leads to an attack rule with a linearly sized right-hand side, and figure 3.11 confirms the resulting exponential runtime.

$$
\begin{aligned}
P_i \cdot S &\overset{\text{c}}{\dashrightarrow} P_{i+1} \cdot S \cdot S && \forall 0 \leq i < n & Q_i \cdot S &\overset{\text{c}}{\dashrightarrow} Q_i \cdot S \cdot S && \forall 0 \leq i < n \\
P_n \cdot S &\overset{\text{i}}{\dashrightarrow} R_n \cdot S && & Q_i \cdot S &\overset{\text{c}}{\dashrightarrow} Q_{i+1} \cdot S \cdot S && \forall 0 \leq i < n \\
R_{i+1} \cdot S &\overset{\text{r}}{\dashrightarrow} R_i && \forall 0 \leq i < n & Q_i \cdot S &\overset{\text{i}}{\dashrightarrow} T_i \cdot S && \forall 0 \leq i < n \\
R_0 \cdot S &\overset{\text{i}}{\dashrightarrow} P_0 \cdot S && \text{if } \neg ref & T_i \cdot S &\overset{\text{r}}{\dashrightarrow} T_i && \forall 0 \leq i < n
\end{aligned}
$$

Figure 3.10: mvPDA with constant local branching

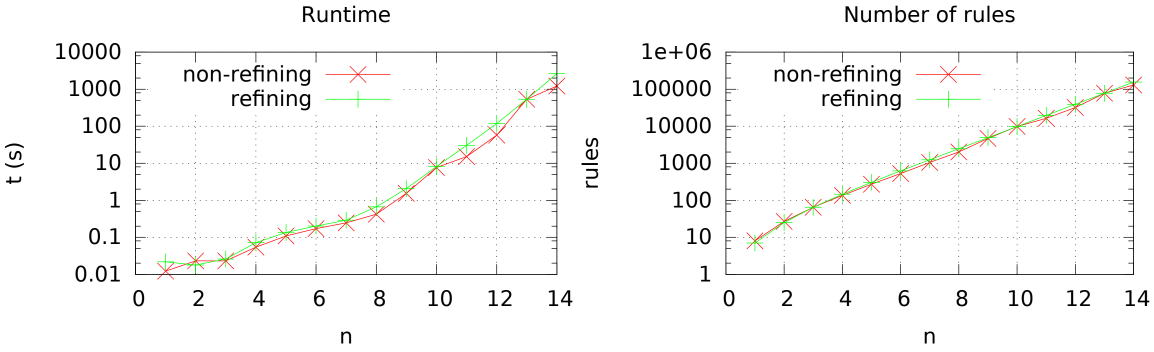

Figure 3.11: Benchmark for mvPDA with constant local branching

Constant global branching For the next family, the global branching degree is restricted, that is the maximum width of any attack tree generated. For that a fixed parameter k is introduced. Figure 3.12 defines a mvPDA as a variation of the mvPDA from figure 3.10, except here there are only $\mathcal{O}(k)$ different states reachable from $Q_0 \cdot S$.

$$\begin{array}{rll rl}
P_i \cdot S \overset{\text{c}}{\dashrightarrow} P_{i+1} \cdot S \cdot S & \forall 0 \leq i < n & \quad & Q_i \cdot S \overset{\text{c}}{\dashrightarrow} Q_i \cdot S \cdot S & \forall 0 \leq i < k \\
P_n \cdot S \overset{\text{i}}{\dashrightarrow} R_n \cdot S & & & Q_i \cdot S \overset{\text{c}}{\dashrightarrow} Q_{i+1} \cdot S \cdot S & \forall 0 \leq i < k \\
R_{i+1} \cdot S \overset{\text{r}}{\dashrightarrow} R_i & \forall 0 \leq i < n & & Q_i \cdot S \overset{\text{i}}{\dashrightarrow} T_i \cdot S & \forall 0 \leq i < k \\
R_0 \cdot S \overset{\text{i}}{\dashrightarrow} P_0 \cdot S & \text{if } \neg ref & & T_i \cdot S \overset{\text{r}}{\dashrightarrow} T_i & \forall 0 \leq i < k
\end{array}$$

Figure 3.12: mvPDA with constant global branching

The results of this benchmark are shown in figure 3.13 for a number of k. As there were no significant differences between the refining and non-refining version, only the refining one is shown. For this family, the runtime is linear with regard to n, while the multiplacative constant is exponential with regard to k.

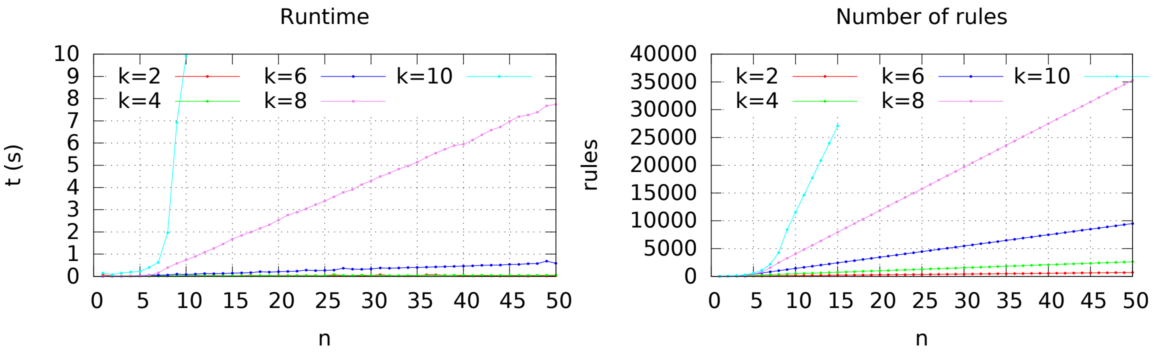

Figure 3.13: Benchmark for mvPDA with constant global branching

Branching in attacking process This family modifies the construction from figure 3.12 and add branching in the attacking process, while keeping branching in the defending process constant. The resulting construction is shown in figure 3.14.

$$
\begin{array}{rll}
P_i \cdot S \overset{\mathrm{c}}{\dashrightarrow} P_i \cdot S \cdot S & \forall 0 \leq i < n \\
P_i \cdot S \overset{\mathrm{c}}{\dashrightarrow} P_{i+1} \cdot S \cdot S & \forall 0 \leq i < n \\
P_n \cdot S \overset{\mathrm{i}}{\dashrightarrow} R_n \cdot S & \\
R_{i+1} \cdot S \overset{\mathrm{r}}{\dashrightarrow} R_i & \forall 0 \leq i < n \\
R_0 \cdot S \overset{\mathrm{i}}{\dashrightarrow} P_0 \cdot S & \text{if } \neg ref
\end{array}
\qquad
\begin{array}{ll}
Q_i \cdot S \overset{\mathrm{c}}{\dashrightarrow} Q_i \cdot S \cdot S & \forall 0 \leq i < k \\
Q_i \cdot S \overset{\mathrm{c}}{\dashrightarrow} Q_{i+1} \cdot S \cdot S & \forall 0 \leq i < k \\
Q_i \cdot S \overset{\mathrm{i}}{\dashrightarrow} T_i \cdot S & \forall 0 \leq i < k \\
T_i \cdot S \overset{\mathrm{r}}{\dashrightarrow} T_i & \forall 0 \leq i < k
\end{array}
$$

Figure 3.14: mvPDA with branching in attacking process

The results of this benchmark are shown in figure 3.15. Surprisingly, the runtime is polynomial when $k \leq 3$ and exponential for $k > 3$. The exponential growth comes from a combination of branching rules from both sides, leading to attack rules with a right-hand of size in $\mathcal{O}(nk)$ and runtime in $2^{\mathcal{O}(nk)}$. For $k \leq 3$, some analysis showed that the combination heuristic could avoid the exponential growth.

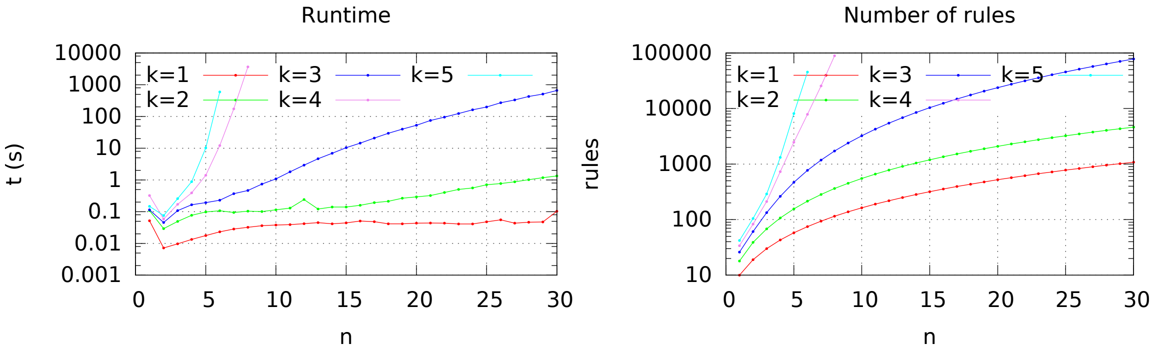

Figure 3.15: Benchmark for mvPDA with branching in attacking process

Discussion The results from the benchmarks show that branching by non-determinism, either on the attacking and defending side, can lead to exponential runtime in $2^{\Omega(n)}$. This can happen even if the local branching degree is constant. Branching on both sides even leads to a runtime in $2^{\Omega(n^2)}$. If the global branching can be kept constant, however, the runtime is polynomial.

Further, different orders in which rules are combined can also cause a change in the runtime. While these are dependant on the input, there still could be a better heuristic that works better on most inputs. However, this should be tested on real-world data, of which there is not much available so far.

4 Conclusion

4.1 Main results

In this thesis, an algorithm for deciding modal refinement between modal visibly pushdown automata was presented. This was done by use of modal process rewrite systems and a new representation of refinement by attack trees.

The theoretical part introduced the concepts and showed correctness of the construction. The practical part described the implementation, including some optimizations, and analyzed the complexity with theoretical bounds and a performance evaluation. This showed the unavoidable exponential runtime and conditions on input which produce or avoid it.

This will hopefully contribute to the usage of mvPDA in the fields applying modal transitions systems, such as in system specifications and model checking, by providing a sound and complete way to test refinement.

4.2 Further extensions

Even with the exponential runtime, there is certainly still room for more optimizations to improve the runtime. One possibility would be deciding positive refinement in certain cases before exploring the whole rule space, for example by showing that there are loops from each attack transition. It should also be possible reduce the worst-time complexity to $2^{\mathcal{O}(n^2)}$ instead of $2^{\mathcal{O}(n^6)}$.

Further algorithms to decide refinement between other classes of mPRS could be developed, at least where it is possible, for example between finite mFSM and general mPRS [BK12]. Also there could be a specialized algorithm for mvBPA, for which the refinement is PTIME-complete.

The tool also still needs to be integrated into the MoTraS system to be made available for end users. Then more options could be added, for example more verbose output such as optionally providing a counterexample in case of negative refinement.

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.

[Wal96] I. Walukiewicz, *Pushdown processes: games and model checking*, Lecture Notes in Computer Science **1102** (1996), 62--75.