THREAD ALGEBRA AND RISK ASSESSMENT SERVICES

JAN A. BERGSTRA, INGE BETHKE, AND ALBAN PONSE

Abstract. Threads as contained in a thread algebra emerge from the behavioral abstraction from programs in an appropriate program algebra. Threads may make use of services such as stacks, and a thread using a single stack is called a pushdown thread. Equivalence of pushdown threads is decidable. Using this decidability result, an alternative to Cohen's impossibility result on virus detection is discussed and some results on risk assessment services are proved.

§1. **Introduction.** This paper is about thread algebra [1, 5]. Threads are processes tailored to describe sequential program behaviour and emerge from the behavioral abstraction of sequential programs. A *basic* thread models a finite program behaviour to be controlled by some execution environment: upon each action (e.g., a request for some service), a reply `true` or `false` from the environment determines further execution. Any execution trace of a basic thread ends either in the (successful) termination state or in the deadlock state. Both these states are modeled as special thread constants. *Regular* threads extend basic threads by comprising loop behaviour, and are reminiscent of flowcharts [14, 12]. Threads may make use of services, i.e., devices that control (part of) their execution by consuming actions, providing the appropriate reply, and suppressing observable activity. Regular threads using the service of a single stack are called *pushdown* threads. Apart from the distinction between deadlock and termination, pushdown threads are comparable to pushdown automata or pushdown processes as described by Stirling [17] or Burkart and Steffen [9].

First, we recall from our companion paper [2] that equivalence of pushdown threads is decidable, and we provide a sketch of our proof. Then we elaborate on Cohen's impossibility result on virus detection [10] (in that 1984 paper, the term *computer virus* was coined). Whereas Cohen showed that a test predicate that decides whether a program executes (and spreads) a virus cannot exist, we proposed in [8] a more modest test that can be used to forecast whether the execution of a thread has no security hazard. This is decidable for regular threads (as argued in [8]), and also for *shrat-safe* pushdown threads (as argued in this paper). In our approach, a security hazard is modeled as the occurrence

Logic Colloquium '05
Edited by C. Dimitracopoulos, L. Newelski, D. Normann, and J. Steel
Lecture Notes in Logic, 28

of a certain action in a thread. We define a service SHRAT (security hazard risk assessment tool) that provides the replies to such tests. The idea is as follows: a security hazard is modeled by an action `risk` and the security hazard risk test as `sh.ok`. In case SHRAT replies `true` to

$$\texttt{if sh.ok then } P \texttt{ else } Q,$$

P will not execute `risk` and execution continues with P. In the other case (reply `false`), Q will be executed instead because P would execute `risk` (there is no security hazard risk assessment of Q). A major point is whether P itself may or may not execute `sh.ok` tests. If P is regular, this is not a problem and we prove that SHRAT is correct. In the case that P is a pushdown thread, correctness only follows if P is *shrat-safe*, i.e., contains no occurrences of both `sh.ok` and `risk` (this is a decidable property).

Our approach offers an alternative to that of Cohen in his well-known paper [10] which shows the impossibility of a test action that reacts on two arguments P and Q at the same time. More precisely, Cohen considers a decision procedure `D` (a predicate on program texts) that determines whether a program executes (and spreads) a virus. Then Cohen's impossibility result is established by the program `C` defined by

$$\texttt{C} = \texttt{if } \neg\texttt{D(C) then } P \texttt{ else } Q,$$

where P executes a virus, and Q is virus-free.

§2. Threads and services. In this section we recall the definitions of basic threads and regular threads. Furthermore we discuss services that may be used by a thread, and we consider the use-operator, which defines how a thread uses a service.

2.1. Threads. *Basic thread algebra* [5][1], BTA, is tailored for the description of sequential program behaviour. Based on a finite set of *actions* A, it has the following constants and operators:

- the *termination* constant S,
- the *deadlock* or *inaction* constant D,
- for each $a \in A$, a binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_$.

We use *action prefixing* $a \circ P$ as an abbreviation for $P \trianglelefteq a \trianglerighteq P$ and take $\circ$ to bind strongest.

The operational intuition behind thread algebra is that each action represents a command which is to be processed by the execution environment of a thread. More specifically, an action is taken as a command for a service offered by the environment. The processing of a command may involve a change of state of this environment. At completion of the processing of the command, the service concerned produces a reply value `true` or `false` to the

[1] In [4], basic thread algebra is introduced under the name *basic polarized process algebra*.

thread under execution. The thread $P \trianglelefteq a \trianglerighteq Q$ will then proceed as P if the processing of a yielded the reply `true` indicating successful processing, and it will proceed as Q if the processing of a yielded the reply `false`.

BTA can be equipped with a partial order and an *approximation operator* in the following way:

1. $\sqsubseteq$ is the partial ordering on BTA generated by the clauses
   (a) for all $P \in \mathrm{BTA}$, $\mathsf{D} \sqsubseteq P$, and
   (b) for all $P_1, P_2, Q_1, Q_2 \in \mathrm{BTA}$, $a \in A$,

$$P_1 \sqsubseteq Q_1 \,\&\, P_2 \sqsubseteq Q_2 \Rightarrow P_1 \trianglelefteq a \trianglerighteq P_2 \sqsubseteq Q_1 \trianglelefteq a \trianglerighteq Q_2.$$

2. $\pi : \mathbb{N} \times \mathrm{BTA} \to \mathrm{BTA}$ is the approximation operator determined by the equations
   (a) for all $P \in \mathrm{BTA}$, $\pi(0, P) = \mathsf{D}$,
   (b) for all $n \in \mathbb{N}$, $\pi(n+1, \mathsf{S}) = \mathsf{S}$, $\pi(n+1, \mathsf{D}) = \mathsf{D}$, and
   (c) for all $P, Q \in \mathrm{BTA}$, $n \in \mathbb{N}$,

$$\pi(n+1, P \trianglelefteq a \trianglerighteq Q) = \pi(n, P) \trianglelefteq a \trianglerighteq \pi(n, Q).$$

We further write $\pi_n(P)$ instead of $\pi(n, P)$.

The operator π finitely approximates every thread in BTA. That is, for all $P \in \mathrm{BTA}$,

$$\exists n \in \mathbb{N}\ \pi_0(P) \sqsubseteq \pi_1(P) \sqsubseteq \cdots \sqsubseteq \pi_n(P) = \pi_{n+1}(P) = \cdots = P.$$

Every thread in BTA is finite in the sense that there is a finite upper bound to the number of consecutive actions it can perform. Following the metric theory of [11] in the form developed as the basis of the introduction of processes in [3], BTA has a completion BTA^∞ which comprises also the infinite threads. Standard properties of the completion technique yield that we may take BTA^∞ as the cpo consisting of all so-called *projective* sequences. That is,

$$\mathrm{BTA}^\infty = \{(P_n)_{n\in\mathbb{N}} \mid \forall n \in \mathbb{N}\ (P_n \in \mathrm{BTA} \,\&\, \pi_n(P_{n+1}) = P_n)\}$$

with

$$(P_n)_{n\in\mathbb{N}} \sqsubseteq (Q_n)_{n\in\mathbb{N}} \Leftrightarrow \forall n \in \mathbb{N}\ P_n \sqsubseteq Q_n$$

and

$$(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}} \Leftrightarrow \forall n \in \mathbb{N}\ P_n = Q_n.$$

For a detailed account of this construction see [1]. In this cpo structure, finite linear recursive specifications represent continuous operators having as unique fixed points *regular* threads, i.e., threads which can only reach finitely many states. A finite linear recursive specification over BTA is a set of equations

$$X_i = t_i(\overline{X})$$

for $i \in I$ with I some finite index set and all $t_i(\overline{X})$ of the form S, D, or $X_{i_l} \trianglelefteq a_i \trianglerighteq X_{i_r}$ for $i_l, i_r \in I$.

EXAMPLE 2.1.1. We define the regular threads

1. $a \circ b \circ \mathsf{D}$,
2. $a \circ b \circ \mathsf{S}$ and
3. $(a \circ b)^\infty$ (this informal notation is explained below)

as the fixed points for X_1 in the specifications

1. $X_1 = a \circ X_2, X_2 = b \circ X_3, X_3 = \mathsf{D}$,
2. $X_1 = a \circ X_2, X_2 = b \circ X_3, X_3 = \mathsf{S}$,
3. $X_1 = a \circ X_2, X_2 = b \circ X_1$, respectively.

Both $a \circ b \circ \mathsf{D}$ and $a \circ b \circ \mathsf{S}$ are finite threads; $(a \circ b)^\infty$ is the infinite thread corresponding to the projective sequence $(P_n)_{n\in\mathbb{N}}$ with $P_0 = \mathsf{D}$, $P_1 = a \circ \mathsf{D}$ and $P_{n+2} = a \circ (b \circ P_n)$. Observe that $a \circ b \circ \mathsf{D} \sqsubseteq a \circ b \circ \mathsf{S}$, $a \circ b \circ \mathsf{D} \sqsubseteq (a \circ b)^\infty$, but $a \circ b \circ \mathsf{S} \not\sqsubseteq (a \circ b)^\infty$.

CONVENTION 2.1.2. In reasoning with finite linear recursive specifications, we shall from now on identify variables and their fixed points. For example, we say that P *is* the regular thread defined by $P = a \circ P$ instead of stating that P equals the fixed point for X in $X = a \circ X$.

2.2. Services. A *service* is a component of an execution architecture for threads that can be used to determine the reply to an action. In [6] various services (called *state machines* in that paper) were considered, as well as their possible role in thread execution. A service is a pair $\langle \Sigma, F \rangle$ consisting of a set Σ of so-called *co-actions* and a reply function F. The reply function F of a service $\langle \Sigma, F \rangle$ is a mapping that gives for each sequence of co-actions in Σ^+ the reply produced by the service. This reply is a boolean value `true` or `false`.

EXAMPLE 2.2.1 (*Stack*). One of the services that will occur in what follows is the *stack* $S = \langle \Sigma, F \rangle$ with $\Sigma = \{\mathtt{push}{:}i, \mathtt{topeq}{:}i, \mathtt{empty}, \mathtt{pop} \mid i \in I\}$ for some finite set I, where $\mathtt{push}{:}i$ pushes i onto the stack and yields reply `true`, the action $\mathtt{topeq}{:}i$ tests whether i is on top of the stack, `empty` tests whether the stack is empty, and `pop` pops the stack if it is non-empty with reply `true` and yields the reply `false` otherwise (leaving the stack empty). By $S(\alpha)$ we denote a stack with contents $\alpha \in I^*$ with the leftmost element of α on top in case $\alpha \neq \varepsilon$ with ε the empty stack contents. In Example 3.1.1 we return to the use of a stack as a service.

In order to provide a specific description of the interaction between a thread and a service, we will use for actions the general notation `c.a` where `c` is the so-called *channel* or *focus* and `a` is a co-action. For example, we write `s.pop` to denote the action which pops a stack via channel `s`.

For a service $\mathcal{S} = \langle \Sigma, F \rangle$ and a finite thread P, we define P *using* the service $\mathcal{S}$ via channel c, notation $P/_{\mathsf{c}}\, \mathcal{S}$, by the following rules:

$$\begin{array}{rcl} \mathsf{S}/_{\mathsf{c}}\, \mathcal{S} & = & \mathsf{S}, \\ \mathsf{D}/_{\mathsf{c}}\, \mathcal{S} & = & \mathsf{D}, \\ (P \trianglelefteq \mathsf{c}'.\mathsf{a} \trianglerighteq Q)/_{\mathsf{c}}\, \mathcal{S} & = & (P/_{\mathsf{c}}\, \mathcal{S}) \trianglelefteq \mathsf{c}'.\mathsf{a} \trianglerighteq (Q/_{\mathsf{c}}\, \mathcal{S}) \text{ if } \mathsf{c}' \neq \mathsf{c}, \\ (P \trianglelefteq \mathsf{c}.\mathsf{a} \trianglerighteq Q)/_{\mathsf{c}}\, \mathcal{S} & = & P/_{\mathsf{c}}\, \mathcal{S}' \text{ if } \mathsf{a} \in \Sigma \text{ and } F(\mathsf{a}) = \mathtt{true}, \\ (P \trianglelefteq \mathsf{c}.\mathsf{a} \trianglerighteq Q)/_{\mathsf{c}}\, \mathcal{S} & = & Q/_{\mathsf{c}}\, \mathcal{S}' \text{ if } \mathsf{a} \in \Sigma \text{ and } F(\mathsf{a}) = \mathtt{false}, \\ (P \trianglelefteq \mathsf{c}.\mathsf{a} \trianglerighteq Q)/_{\mathsf{c}}\, \mathcal{S} & = & \mathsf{D} \text{ if } \mathsf{a} \notin \Sigma, \end{array}$$

where $\mathcal{S}' = \langle \Sigma, F' \rangle$ with $F'(\sigma) = F(\mathsf{a}\sigma)$ for all co-action sequences $\sigma \in \Sigma^+$. Note that actions that use a service $\mathcal{S}$ are not observable. The use operator is expanded to infinite threads P by stipulating

$$P/_{\mathsf{c}}\, \mathcal{S} = (\pi_n(P)/_{\mathsf{c}}\, \mathcal{S})_{n \in \mathbb{N}}.$$

As a consequence, $P/_{\mathsf{c}}\, \mathcal{S} = \mathsf{D}$ if for every n, $\pi_n(P)/_{\mathsf{c}}\, \mathcal{S} = \mathsf{D}$.

EXAMPLE 2.2.2. We consider again the threads $a \circ b \circ \mathsf{D}$, $a \circ b \circ \mathsf{S}$ and $(a \circ b)^\infty$ from Example 2.1.1 but now in the versions $\mathsf{c.a} \circ \mathsf{c.b} \circ \mathsf{D}$, $\mathsf{c.a} \circ \mathsf{c.b} \circ \mathsf{S}$ and $(\mathsf{c.a} \circ \mathsf{c.b})^\infty$ for some channel c and service $\mathcal{S} = \langle \{\mathsf{a}, \mathsf{b}\}, F \rangle$. Then $(\mathsf{c.a} \circ \mathsf{c.b} \circ \mathsf{D})/_{\mathsf{c}}\, \mathcal{S} = \mathsf{D}$ and $(\mathsf{c.a} \circ \mathsf{c.b} \circ \mathsf{S})/_{\mathsf{c}}\, \mathcal{S} = \mathsf{S}$, but $(\mathsf{c.a} \circ \mathsf{c.b})^\infty/_{\mathsf{c}}\, \mathcal{S} = \mathsf{D}$.

§3. Pushdown threads and decidable equivalence.

In this section we consider pushdown threads, i.e., regular threads that use a stack. Then, we recall from our paper [2] that equivalence of pushdown threads is decidable and sketch a proof of this fact.

3.1. Pushdown threads.

In the next example we show that the use of services may turn regular threads into non-regular ones.

EXAMPLE 3.1.1. Let $\{a, b, \mathtt{s.push{:}1}, \mathtt{s.pop}\} \subseteq A$, where the last two actions refer to the stack S defined in Example 2.2.1 with $I = \{1\}$. By the defining equations for the use operator it follows that for any thread P and $\sigma \in \{1\}^*$,

$$(\mathtt{s.push{:}1} \circ P)/_{\mathsf{s}}\, S(\sigma) = P/_{\mathsf{s}}\, S(1\sigma).$$

Furthermore, it easily follows that

$$(P \trianglelefteq \mathsf{s.pop} \trianglerighteq \mathsf{S})/_{\mathsf{s}}\, S(\sigma) = \begin{cases} \mathsf{S} & \text{if } \sigma = \varepsilon \text{ (the empty sequence)}, \\ P/_{\mathsf{s}}\, S(\rho) & \text{if } \sigma = 1\rho. \end{cases}$$

Now consider the regular thread Q defined by [2]

$$\begin{aligned} Q &= (\mathtt{s.push{:}1} \circ Q) \trianglelefteq a \trianglerighteq R, \\ R &= b \circ R \trianglelefteq \mathsf{s.pop} \trianglerighteq \mathsf{S}. \end{aligned}$$

[2] Note that a *linear* recursive specification of Q requires (at least) five equations.

Then for all $\sigma \in \{1\}^*$,

$$\begin{aligned} Q/_{\mathsf{S}}\, S(\sigma) &= ((\mathtt{s.push{:}1} \circ Q) \trianglelefteq a \trianglerighteq R)/_{\mathsf{S}}\, S(\sigma) \\ &= (Q/_{\mathsf{S}}\, S(1\sigma)) \trianglelefteq a \trianglerighteq (R/_{\mathsf{S}}\, S(\sigma)), \\ R/_{\mathsf{S}}\, S(1\sigma) &= b \circ R/_{\mathsf{S}}\, S(\sigma), \\ R/_{\mathsf{S}}\, S(\varepsilon) &= \mathsf{S}. \end{aligned}$$

It is not hard to see that $Q/_{\mathsf{S}}\, S(\varepsilon)$ is an infinite thread with the property that for all $n \in \mathbb{N}$, a trace of $n{+}1$ a-actions produced by n positive and one negative reply on a is followed by n b-actions and S. This yields an *non-regular* thread: if $Q/_{\mathsf{S}}\, S(\varepsilon)$ were regular, it would be a fixed point of some finite linear recursive specification, say with k equations. But specifying a trace containing k b-actions followed by S already requires $k{+}1$ linear equations $X_1 = b \circ X_2, \ldots, X_k = b \circ X_{k+1}, X_{k+1} = \mathsf{S}$, which contradicts the assumption. So $Q/_{\mathsf{S}}\, S(\varepsilon)$ is not regular.

We call a regular thread that uses a stack as described in Example 2.2.1 a *pushdown thread*. In what follows we assume that pushdown threads are given with help of a distinguished identifier from a finite linear recursive specification $\mathcal{F}$ and a stack over some fixed alphabet. The equations in $\mathcal{F}$ may contain actions that address the stack via the use-application $/_{\mathsf{S}}$.

3.2. Decidable equivalence. From our companion paper [2] we quote the following result:

THEOREM 3.2.1. *Equivalence of pushdown threads is decidable.*

This theorem follows from a reduction to the dpda-equivalence problem whose decidability was proved by Sénizergues [15, 16]. Here we provide only a sketch, a detailed proof can be found in [2].

The idea is to use a transformation from pushdown threads to dpda's such that the identity

$$P/_{\mathsf{S}}\, S(\alpha) = Q/_{\mathsf{S}}\, S(\beta)$$

holds if and only if the identity

$$L(\mathcal{A}, P'\alpha') = L(\mathcal{A}, Q'\beta')$$

holds, where the latter identity expresses that for the derived dpda $\mathcal{A}$, the language accepted by 'configuration' $P'\alpha'$ equals the one accepted by configuration $Q'\beta'$. The transformation described in [2] consists of five steps and uses the dpda-equivalence result as formulated by Stirling [18] because this is closer to our setting:

1. Transform $P/_{\mathsf{S}}\, S(\alpha)$ and $Q/_{\mathsf{S}}\, S(\beta)$ such that initially the stacks are non-empty (also if one of α and β is the empty string), and such that upon their termination the stack is empty. The reason for this step stems from the fact that language acceptance for dpda's is defined on configurations

of the form $R\alpha$ where R is a 'state' and α is a non-empty stack contents. A word w is in the accepted language iff the dpda in initial state R empties the stack by performing the transitions whose labels form w.

2. Replace occurrences of D by loops that fill the stack (e.g., replace $P_i = \mathsf{D}$ by $P_i = \mathtt{s.push}\!:\! j \circ P_i$ for some $j \in I$). The reason for this step is that D has no equivalent in the dpda-equivalence result.
3. Normalize infinite traces: replace each equation $P_i = P_l \trianglelefteq a \trianglerighteq P_r$ by $P_i = \overline{\mathsf{S}} \trianglelefteq b \trianglerighteq (P_l \trianglelefteq a \trianglerighteq P_r)$ with b an action that occurs not in P and Q. Here $\overline{\mathsf{S}}$ is the thread that first empties the stack and then terminates ($\overline{S}$ is also used in step 1). The reason for this step is that each infinite trace becomes interlarded with exits b, and is thus characterized by finite traces which in turn are subject to dpda language acceptance.
4. Construction of an associated pushdown automaton (pda). The specifications of the so far transformed $P(\alpha)$ and $Q(\beta)$ admit a straightforward definition of a pda whose transitions are deterministic. The only remaining problem is that the ε-transitions (that stem from stack actions) need not pop the stack, as required by the decidability result in [18].
5. Construction of a dpda in which the ε-transitions only pop the stack. The pda thus obtained is transformed by changing its transition rules for ε. Those that do not pop the stack are either swallowed by an observable transition and yield a new transition rule, or form a loop, in which case they can be omitted. This step preserves language acceptance and concludes the transformation.

We will exploit this decidability result by replacing certain equations in the definition of the regular thread that underlies a pushdown thread, i.e. in the definition of P when considering $P/_{\mathtt{s}}\, S(\alpha)$. For example, it is decidable whether a pushdown thread is *normed*, i.e., has the option to terminate (to end in S): let a linear recursive specification

$$\mathcal{F} = \{P_i = t_i(\vec{P}) \mid i = 1, \ldots, n\}$$

be given (and thus a repertoire of stack actions and external actions). Replace each equation $P_i = \mathsf{S} \in \mathcal{F}$ by $\overline{P}_i = a \circ \overline{P}_i$ and overline all remaining identifiers. Then $P_k/_{\mathtt{s}}\, S(\alpha)$ is normed $\Leftrightarrow P_k/_{\mathtt{s}}\, S(\alpha) \neq \overline{P}_k/_{\mathtt{s}}\, S(\alpha)$.

REMARK 3.2.2. Interestingly, inclusion of pushdown threads is not decidable (although two pushdown threads are equivalent if they are included in each other). This follows from a reduction to the halting problem for Minsky machines — an approach also taken in Jančar et al. [13]. A detailed proof is recorded in [2].

§4. Security hazard risk assessment. In this section we consider the possibility that a pushdown thread uses a service that supports forecasting of certain future behaviour. In [7] various such services are studied (e.g., the

halting problem and "rational agents") and in [8] we discuss a rather specific case: a service SHRAT (*security hazard risk assessment tool*). In this paper we provide a detailed construction of SHRAT for regular threads and a proof of its correctness. Finally, we consider SHRAT for pushdown processes and distinguish the case of shrat-safe threads.

4.1. A definition of SHRAT. We model a security hazard in a pushdown thread $\mathcal{P}$ as the execution of an action `risk`. Furthermore, $\mathcal{P}$ may contain a test action `sh.ok` that can use the service SHRAT to forecast whether `risk` will be executed: SHRAT replies `true` to

$$\mathcal{Q} \trianglelefteq \texttt{sh.ok} \trianglerighteq \mathcal{R}$$

if $\mathcal{Q}$ does not execute `risk`, and `false` if $\mathcal{Q}$ does execute the action `risk` (and then $\mathcal{R}$ is executed instead). In order to model forecasting, we first define the *residual thread* of a pushdown thread $\mathcal{P}$ as the thread that remains after zero or more actions of $\mathcal{P}$ have been executed:

DEFINITION 4.1.1. Let $\mathcal{P}$ be a pushdown thread. We write $\mathcal{Q} \in \mathit{Res}(\mathcal{P})$ whenever $\mathcal{Q}$ is a *residual thread* of $\mathcal{P}$:

- $\mathcal{P} \in \mathit{Res}(\mathcal{P})$,
- $\mathcal{P} \in \mathit{Res}(\mathcal{P} \trianglelefteq a \trianglerighteq \mathcal{Q})$,
- $\mathcal{Q} \in \mathit{Res}(\mathcal{P} \trianglelefteq a \trianglerighteq \mathcal{Q})$, and
- if $\mathcal{R} \in \mathit{Res}(\mathcal{Q})$ and $\mathcal{Q} \in \mathit{Res}(\mathcal{P})$, then $\mathcal{R} \in \mathit{Res}(\mathcal{P})$.

Of course, the very idea of a service SHRAT that supports forecasting of the execution of future actions `risk` in a residual thread $\mathcal{Q} \trianglelefteq \texttt{sh.ok} \trianglerighteq \mathcal{R}$ of $\mathcal{P}$, thus

$$(\mathcal{Q} \trianglelefteq \texttt{sh.ok} \trianglerighteq \mathcal{R})/_{\texttt{sh}}\ \mathrm{SHRAT} \tag{1}$$

requires that SHRAT is aware of the specification of $\mathcal{Q}$. So, a reply function that only uses the current co-action and those processed before is in this case not sufficient. It seems most natural to model that SHRAT "gets to know and analyzes" $\mathcal{Q}$'s specification upon the request `sh.ok` in the use-application (1) above. We describe this change of state of SHRAT and the resulting reply in the following definition.

DEFINITION 4.1.2. Let a pushdown thread $\mathcal{P}$ be given by some specification $\mathcal{F}_\mathcal{P}$ and let `sh.ok` be the only action in $\mathcal{P}$ with focus `sh`. Then the service SHRAT is defined by the following two properties:

(1) for any residual thread $\mathcal{Q} \trianglelefteq \texttt{sh.ok} \trianglerighteq \mathcal{R}$ of $\mathcal{P}$,

$$(\mathcal{Q} \trianglelefteq \texttt{sh.ok} \trianglerighteq \mathcal{R})/_{\texttt{sh}}\ \mathrm{SHRAT} = (\mathcal{Q} \trianglelefteq \texttt{sh.ok} \trianglerighteq \mathcal{R})/_{\texttt{sh}}\ \mathrm{SHRAT}(\mathcal{F}_\mathcal{P}, \mathcal{Q}),$$

where $\mathrm{SHRAT}(\mathcal{F}_\mathcal{P}, \mathcal{Q})$ is the instance of SHRAT that has loaded $\mathcal{F}_\mathcal{P}$ and analyzed $\mathcal{Q}$, and

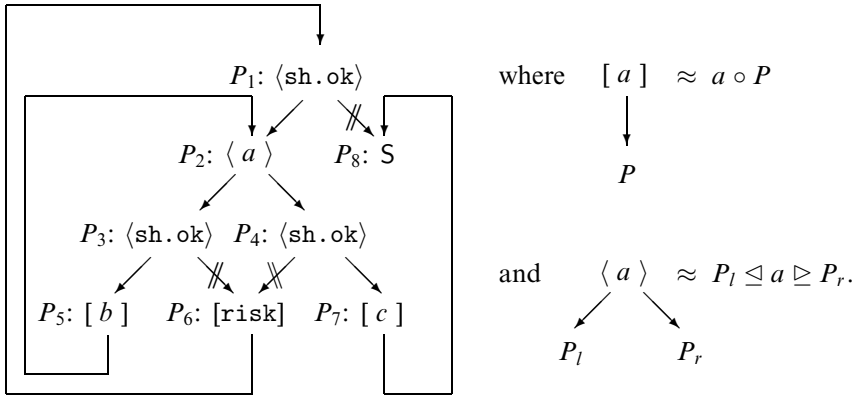
(2) $(\mathcal{Q} \trianglelefteq \mathtt{sh.ok} \trianglerighteq \mathcal{R})/_{\mathtt{sh}}\,\mathrm{SHRAT}(\mathcal{F}_\mathcal{P}, \mathcal{Q}) =$

$$\begin{cases} \mathcal{Q}/_{\mathtt{sh}}\,\mathrm{SHRAT} & \text{(thus reply } \mathtt{true}\text{) if no } \mathtt{risk}\text{-action} \\ & \text{will be executed in } \mathcal{Q}/_{\mathtt{sh}}\,\mathrm{SHRAT}, \\ \mathcal{R}/_{\mathtt{sh}}\,\mathrm{SHRAT} & \text{(thus reply } \mathtt{false}\text{) if a } \mathtt{risk}\text{-action} \\ & \text{will be executed in } \mathcal{Q}/_{\mathtt{sh}}\,\mathrm{SHRAT}. \end{cases}$$

The (instantiated) service $\mathrm{SHRAT}(\mathcal{F}_\mathcal{P}, \mathcal{Q})$ models a "security hazard risk assessment" in the sense that if a security hazard in $\mathcal{Q}$ is modeled by the execution of the action $\mathtt{risk}$, the reply $\mathtt{true}$ to $\mathcal{Q} \trianglelefteq \mathtt{sh.ok} \trianglerighteq \mathcal{R}$ ensures that in the residual thread $\mathcal{Q}/_{\mathtt{sh}}\,\mathrm{SHRAT}$ no security hazard will occur (cf. [8]).

It can be the case that $\mathrm{SHRAT}(\mathcal{F}_\mathcal{P}, \mathcal{Q})$ replies $\mathtt{true}$ because SHRAT will reply $\mathtt{false}$ to a future $\mathtt{sh.ok}$-test in $\mathcal{Q}/_{\mathtt{sh}}\mathrm{SHRAT}$. For example, in the regular thread P_1 given and depicted below, the various $\mathtt{sh.ok}$-tests are evaluated as follows:

$$\begin{array}{lcll} P_1 & = & P_2 \trianglelefteq \mathtt{sh.ok} \trianglerighteq P_8 & (\mathtt{true}) \\ P_2 & = & P_3 \trianglelefteq a \trianglerighteq P_4 & \\ P_3 & = & P_5 \trianglelefteq \mathtt{sh.ok} \trianglerighteq P_6 & (\mathtt{true}) \\ P_4 & = & P_6 \trianglelefteq \mathtt{sh.ok} \trianglerighteq P_7 & (\mathtt{false}) \end{array} \qquad \begin{array}{lcl} P_5 & = & b \circ P_2 \\ P_6 & = & \mathtt{risk} \circ P_1 \\ P_7 & = & c \circ P_8 \\ P_8 & = & \mathsf{S}. \end{array}$$



Clearly, the thread $\mathcal{T} = P_1/_{\mathtt{sh}}\,\mathrm{SHRAT}$ satisfies $\mathcal{T} = b \circ \mathcal{T} \trianglelefteq a \trianglerighteq c \circ \mathsf{S}$.

In the next section we discuss how to instantiate SHRAT for regular threads in an appropriate way.

4.2. SHRAT for regular threads. Following Convention 2.1.2, we assume that if a regular thread P_1 is given, it is given by a linear recursive specification $\mathcal{F}_{P_1}$ that contains an equation $P_1 = t_1(\vec{P})$. Furthermore, we say that an equation $P_j = P_l \trianglelefteq a \trianglerighteq P_r$ in $\mathcal{F}_{P_1}$ has a *predecessor* if P_j occurs in the right-hand side of at least one equation. Finally, we restrict to specifications $\mathcal{F}_{P_1}$

with the property that if $P_j = P_l \trianglelefteq \mathtt{sh.ok} \trianglerighteq P_r \in \mathcal{F}_{P_1}$, then $l \neq r$ (otherwise, the reply to `sh.ok` would be meaningless).

Starting from $P_1/_{\mathtt{sh}}$ SHRAT with the regular thread P_1 specified in $\mathcal{F}_{P_1}$, we provide an algorithm that upon each residual thread of the form

$$(P_m \trianglelefteq \mathtt{sh.ok} \trianglerighteq P_j)/_{\mathtt{sh}} \text{ SHRAT}$$

constructs an instantiated service $\text{SHRAT}(\mathcal{F}_{P_1}, P_m)$ that gives the correct reply. Typical for this algorithm is that $\text{SHRAT}(\mathcal{F}_{P_1}, P_m)$ contains a copy of $\mathcal{F}_{P_1}$ in which *all* `sh.ok` actions are annotated with the correct reply. To this end, $\mathcal{F}_{P_1}$ is loaded into SHRAT and analyzed as follows: number each equation that contains a `risk`-occurrence starting from 1. Then, for each numbered equation label each predecessor equation with the next free number until a connecting `sh.ok`-equation is found, or a loop occurs, or an equation without predecessors is found. In the case that some `sh.ok`-equation is found and connects via its `true`-branch, its `sh.ok`-action is annotated `false` ($\mathtt{sh.ok}^{\mathtt{false}}$); if it connects via its `false`-branch, the equation is labeled with a fresh negative number (it may *possibly* lead to a `risk`-action, namely when a `false`-annotation is added in a future inspection). Then this procedure is repeated for equations labeled with a negative number, again instantiating first occurrences of `sh.ok`-actions with `false` if their `true`-branch leads to an action `risk`. Finally, all non-annotated `sh.ok`-actions are annotated `true` because their `true`-branch does not lead to a `risk`-action.

In Figure 1, we illustrate how the annotation proceeds: first the two lowest `sh.ok` actions are annotated `false`, and because of the $\searrow$ arrow, the equation of the leftmost one is labeled with a fresh negative number. The combination of the `false`-annotation and this label leads to the `false`-annotation of the topmost `sh.ok`-action.

Construction of $\text{SHRAT}(\mathcal{F}_{P_1}, P_m)$ *for a regular thread* P_1. Let $\mathcal{F}_{P_1} = \{P_i = t_i(\vec{P}) \mid i = 1, \ldots, n\}$ be a linear specification of the regular thread P_1. Upon a residual thread

$$P_m \trianglelefteq \mathtt{sh.ok} \trianglerighteq P_w,$$

the service $\text{SHRAT}(\mathcal{F}_{P_1}, P_m)$ is constructed as follows: load $\mathcal{F}_{P_1}$ in SHRAT. We further call this copy $\mathcal{F}^{\text{an}}_{P_1}$. Label each equation in $\mathcal{F}^{\text{an}}_{P_1}$ that contains `risk` in the right-hand side with a number, starting from 1, say $1, \ldots, k$. If no `risk`-actions occur in $\mathcal{F}^{\text{an}}_{P_1}$, then apply step 3 below. In the other case, apply step 1:

1. On $\mathcal{F}^{\text{an}}_{P_1}$ apply the procedure $Eval^+(1)$, where $Eval^+(i)$ for $i \geq 1$ is defined as follows:

   $Eval^+(i)$: If the equation labeled with number i has the form

   $$(i)\ P_j = P_l \trianglelefteq a \trianglerighteq P_r,$$

   then *evaluate* all P_j occurrences in the right-hand sides of all equations,