Recursive Advice for Coordination *

Michał Terepeta, Hanne Riis Nielson, and Flemming Nielson

Technical University of Denmark
{mtte,riis,nielson}@imm.dtu.dk

Abstract. *Aspect-oriented programming* is a programming paradigm that is often praised for the ability to create modular software and separate cross-cutting concerns. Recently aspects have been also considered in the context of coordination languages, offering similar advantages. However, introducing aspects makes analyzing such languages more difficult due to the fact that aspects can be recursive — *advice* from an aspect must itself be analyzed by aspects — as well as being simultaneously applicable in concurrent threads. Therefore the problem of reachability of various states of a system becomes much more challenging. This is important since ensuring that a system does not contain errors is often equivalent to proving that some states are not reachable.
In this paper we show how to solve these challenges by applying a successful technique from the area of software model checking, namely *communicating pushdown systems*. Even though primarily used for analysis of recursive programs, we are able to adapt them to fit this new context.

1 Introduction

Motivation. Aspect-oriented programming [1,2] is a successful programming paradigm that is used in many environments and supported by all major programming languages. Its biggest advantage is the ability to separate cross-cutting concerns and thus make it possible to create very modular software. The classical example is of course logging — in order to be useful it should be performed in many different and unrelated parts of the code. Aspects allow to separate the code for logging from the code implementing the program logic.

Usually an aspect consists of a pointcut and an advice. A pointcut is basically a pattern that specifies when some join point (e.g. location in a program, some action, etc.) matches the aspect. An advice consists of some additional code or actions that should be executed if the pointcut matches. Often an advice might specify actions that should be performed after, before or instead of the matched one. All of this allows one to easily separate the code for the actual functionality from the code for e.g. logging, which could be specified using aspects.

Aspects have also been used in the context of process calculi — [3] introduced a language called AspectK, which is an extension of the coordination language KLAIM with aspect-oriented features. However, the introduction of

* The research presented in this paper has been supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology.

aspects makes the analysis of such languages much more challenging. When one allows the advice to contain before and/or after actions, the code can suddenly "grow" due to advice. Moreover the actions from the advice must itself be subject to other aspects as they have a recursive structure. Therefore it is possible that a process could grow arbitrarily large. In this paper we focus on solving these challenges.

Contributions. We define a process calculi allowing concurrent threads communicating via message-passing. The definition of aspects is quite similar to the one in [3] and allows both before and after actions. Our main contribution is the use of a technique from software model checking field, namely pushdown systems, which have been used for software model checking [4] and static analysis [5,6,7] of recursive programs. We show that they can be adapted to our context and give us the ability to model arbitrarily large processes as well as to summarize the actions they execute. Furthermore, we present how to go a step further and reason about concurrent systems with aspects. We focus on proving unreachability of certain states of such systems since the problem of reachability is often of highest importance — many desired properties of various systems (not only concurrent) can be reduced to the question of reachability of the error states.

Related work. In [8] static analysis techniques have been used to analyze AspectKE (and a programming language AspectKE*), which are based on AspectK. However, these languages do not allow the advice to contain before or after actions, which practically avoids the above mentioned challenges. In this paper we focus on techniques that allow lifting this restriction. In [9,10] the authors use communicating pushdown systems to analyze concurrent programs with recursive procedures and synchronization-based communication. This approach has been also applied to C programs in [11]. Our approach is based on this work, it does however, have some crucial differences. Clearly the context is quite different as we do not deal with procedural programming languages but process calculi. In particular the source of recursion are the aspects and not the procedures. Consequently we shall use the stack in a completely different way. In [9,10,11] the stack is used for storing locations of the programs, whereas we do not even have the concept of location and use the stack to represent the actual process itself (i.e. the action to be executed). In [12] pushdown systems are used to analyze concurrent software, but in a setting of shared memory concurrency. Moreover this approach is under-approximating with respect to control flow, since it performs the analysis under a context bound. That is, it limits the number of possible context switches that the threads can make.

Structure of the paper. The paper is organized as follows. First in Sect. 2 we introduce the language that is used for examples and analysis. Then in Sect. 3 we present the basic theory behind pushdown systems and our adaptation to the context of process calculi with aspects. Furthermore in Sect. 4 we illustrate our approach on an example. Finally we conclude in Sect. 5.

2 Language

Let us introduce the language that we will be working with. It allows multiple threads running concurrently and communication via synchronous message passing.

$$
\begin{array}{lll}
\textit{nets} & N & ::= N_1 || N_2 \mid c :: P \mid c :: \mathbf{Rec}\mathsf{X}\,\mathbf{.}\,P \\
\textit{processes} & P & ::= \textstyle\sum_i a_i.Q_i \\
 & Q & ::= P \mid \mathsf{X} \\
\textit{actions} & a & ::= \mathbf{receive}(\bar{p})@c \mid \mathbf{send}(\bar{t})@c \mid \\
 & & \quad \mathbf{if}(e)\ a \mid \mathbf{break} \mid \mathbf{skip} \\
\textit{tests} & e & ::= t_1 = t_2 \mid t_1 \neq t_2 \mid \mathbf{true} \\
\textit{terms} & t & ::= c \mid x \\
\textit{patterns} & p & ::= t \mid !x
\end{array}
$$

For readability we write constants with an uppercase first letter and variables with a lowercase one. One subtlety about **receive** is that the two actions: $\mathbf{receive}(!x)@\mathsf{N}$ and $\mathbf{receive}(x)@\mathsf{N}$ are quite different — the former will evaluate to itself when ready to execute and will accept any value from the process N and bind it to x. Whereas in the latter case x is an already bound variable, so assuming that it is bound to C the action will evaluate to $\mathbf{receive}(\mathsf{C})@\mathsf{N}$ and thus only accept C from N. Finally the communication is performed in CSP style [13] where the **send** and **receive** actions specify the recipient and sender respectively.

As already mentioned, one of the main features of our language is the presence of aspects. They can be defined as follows.

$$
\begin{array}{lll}
\textit{aspects} & \textit{asp} & ::= A[\textit{cut};\ e] \triangleq \textit{adv} \\
\textit{advice} & \textit{adv} & ::= \textit{as}\ \mathbf{break} \mid \textit{as}\ \mathbf{proceed}\ \textit{as} \\
\textit{action sequence} & \textit{as} & ::= a\,\mathbf{.}\,\textit{as} \mid \epsilon \\
\textit{pointcut} & \textit{cut} & ::= c :: a
\end{array}
$$

Informally the semantics of the aspects says that before executing an action we need to check what aspects apply to it and then combine the advice from them. Checking if an aspect applies to an action amounts to pattern matching against the *cut* and evaluating the applicability condition e associated with the aspect. Note that in case of **receive** action with input, e.g. $\mathbf{receive}(!x)@N$, the condition cannot refer to x — the aspect needs to evaluate before executing the action (it must be able to disallow it). But using x in a condition would require executing the action first, thus we do not allow using x in such situations. One of the important things to emphasize here is that we use the order in which aspects are defined and trap actions based on that order. We will use this fact in our analysis. This also means that changing the order can change the behavior of the system. Furthermore the advice from an aspect (i.e. the right-hand side) is also analyzed by all aspects except for the **proceed**, which corresponds to the original trapped action and should be only analyzed by the aspect next in the order. Therefore a single action can be trapped by many aspects.

Example 1. Consider the following process with an aspect (for simplicity we skip here the processes Q and Log and assume that sending anything to them will always succeed).

$$\mathsf{P} :: \mathbf{send}(\mathsf{Test})@\mathsf{Q}$$
$$\mathsf{A}_1[\mathsf{P} :: \mathbf{send}(a)@q;\ \mathbf{true}] \triangleq \mathbf{send}(a)@\mathsf{Log}\,.\,\mathbf{proceed}$$

The aspect will trap any **send** action of P. However, since the advice will also be analyzed by the aspect it will also trap the **send** action directed to Log. So this example actually demonstrates the possibility of non-termination and a process that can grow infinitely large. Clearly the aspect as defined above is not providing the desired behavior. The correct way to specify it is to restrict what actions should be trapped.

$$\mathsf{A}_1[\mathsf{P} :: \mathbf{send}(a)@q;\ q \neq \mathsf{Log}] \triangleq \mathbf{send}(a)@\mathsf{Log}\,.\,\mathbf{proceed}$$

With this small refinement the aspect will only trap the **send** actions that are not sent to the Log. Thus the system will successfully terminate.

We will now present some more involved example that will be used throughout the paper to demonstrate our approach. Imagine an ATM[1] session — it first receives some credentials from the user and checks the credentials against the information stored on the card. If everything matches it dispenses the cash and informs the bank to deduct the given amount from the account. The following definition models this behavior

$$\begin{aligned}\mathsf{ATM} :: \ & \mathbf{receive}(!\mathit{credentials}, !\mathit{amount})@\mathsf{User}\,.\\ & \mathbf{check}(\mathit{credentials})\,.\\ & \mathbf{send}(\mathit{amount})@\mathsf{User}\,.\\ & \mathbf{send}(\mathit{credentials}, \mathit{amount})@\mathsf{Bank}\end{aligned}$$

where **check** is an internal action that does not involve any communication or synchronization and either executes successfully if the credentials are valid or otherwise terminates the session. This process seems reasonable but we can imagine that in order to increase the security of this solution one could add aspects that actually confirmed the credentials with the bank.

$$\begin{aligned}\mathsf{A}_1[\mathsf{ATM} :: \mathbf{check}(c);\ \mathbf{true}] \triangleq\ & \mathbf{proceed}\,.\,\mathbf{send}(c)@\mathsf{Bank}\,.\\ & \mathbf{receive}(!a)@\mathsf{Bank}\\ \mathsf{A}_2[\mathsf{ATM} :: \mathbf{receive}(!a)@\mathsf{Bank};\ \mathbf{true}] \triangleq\ & \mathbf{proceed}\,.\\ & \mathbf{if}(a = \mathsf{Abort})\ \mathbf{send}(\mathsf{ErrorMessage})@\mathsf{User}\,.\\ & \mathbf{if}(a = \mathsf{Abort})\ \mathbf{break}\,.\end{aligned}$$

The above two aspects make the additional check of credentials with the bank (after checking locally) to improve the security. Obviously we want the ATM session to terminate (with an error message) when this check fails.

[1] Automated Teller Machine

Apart from that we need to define the process modelling the Bank

$$\begin{aligned}\mathsf{Bank} :: (&\mathbf{receive}(!credentials)@\mathsf{ATM}\,.\,\mathbf{send}(\mathsf{Ok})@\mathsf{ATM}\,.\\ &\mathbf{receive}(credentials, !amount)@\mathsf{ATM})\\ +(&\mathbf{receive}(!credentials)@\mathsf{ATM}\,.\,\mathbf{send}(\mathsf{Abort})@\mathsf{ATM})\end{aligned}$$

We do not define the process for User since it does not actually bring anything interesting to the example.

Now having such a system, one of the things that we would like to guarantee is that whenever the bank aborts a transaction, the ATM will not dispense the cash. In other words we want to ensure that both the bank and the ATM have a consistent view on the transaction. To achieve that we will use communicating pushdown systems that are introduced in the following section.

3 Pushdown systems

3.1 Basic concepts of pushdown systems

Pushdown systems are a formalism very close to pushdown automata. The main difference is that pushdown automata define languages, i.e. they have some input alphabet and use a stack to decide whether a word is accepted or not. Pushdown systems do not have an input alphabet and thus have a flavor of a description of a system that executes with an unbounded stack[2]. As such they are very useful for analyzing recursive programs — the stack is used to determine the location in the program along with the return locations of the procedures that were called. Below we give the formal definition of a pushdown system.

Definition 1. *A pushdown system is a four-tuple $\mathcal{P} = (Q, \mathsf{Act}, \Gamma, \Delta)$, where Q is a finite set of control locations (in other words states), Act is a finite set of actions, Γ is a finite stack alphabet and Δ is a finite set of pushdown rules of the form $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$ where $p, p' \in Q$, $\gamma \in \Gamma$, $w \in \Gamma^*$ and $a \in \mathsf{Act}$.*

Most definitions of pushdown systems require that $|w| \leq 2$. This is useful especially when discussing some algorithms that take advantage of this property. Moreover this is not a restriction — we can easily transform any set of rules that does not satisfy this assumption into one that does by adding new rules and some fresh control locations. Therefore, for convenience and clarity of presentation we will not impose this requirement.

One of the reasons behind the success of pushdown systems and why the rules are presented in this way is that they correspond quite closely to how a procedural program executes. For this we need just three kinds of rules that correspond to calling a procedure, returning from a procedure and simply progressing to the next statement:

$$\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle \qquad \langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \epsilon\rangle \qquad \langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\rangle$$

[2] Although if the pushdown rules are annotated with actions, as in this paper, then one could say that they constitute the input alphabet.

Moreover it is worth mentioning that $p, p' \in Q$ are often used to store the global state of a program and $\gamma, \gamma', \gamma'' \in \Gamma$ can be used to track the location of the current statement along all the return locations of the various procedures as well as the local state of procedures.

Apart from that, following [9,10], we also need the concept of a configuration of a pushdown system $\mathcal{P}$, which is a pair $\langle q, w \rangle$ such that $q \in Q$ and $w \in \Gamma^*$. Moreover we say that a set of configurations C is regular if for each $q \in Q$ the language $\{w \in \Gamma^* \mid \langle q, w \rangle \in C\}$ is regular. Having the definition of a configuration we can define the transition relation $\stackrel{a}{\Longrightarrow}$ between configurations — if $\langle p, \gamma \rangle \stackrel{a}{\hookrightarrow} \langle p', w \rangle$ then $\langle p, \gamma s \rangle \stackrel{a}{\Longrightarrow} \langle p', ws \rangle$ for all $s \in \Gamma^*$. Moreover, we also define $\stackrel{a_1 \ldots a_n}{\Longrightarrow}$ to be the reflexive, transitive closure of the above relation. Now we can define the concept of successor configuration. We say that c_s is an immediate successor of c if there exists a such that $c \stackrel{a}{\Longrightarrow} c_s$ and is a successor of c' if there exist $a_1 \ldots a_n$ such that $c' \stackrel{a_1 \ldots a_n}{\Longrightarrow} c_s$. Finally the set of all successors of some set of configurations C is defined by:

$$Post^*(C) = \{c' \mid \exists c \in C : c' \text{ is a successor of } c\}$$

The concepts of immediate predecessor, predecessor and set of predecessors are defined analogously.

Clearly a set of pushdown rules gives rise to a possibly infinite transition system and thus a possibly infinite set of reachable configurations. The reason for this is the fact that the stack is unbounded. However, one of the crucial results in this area is that the set of successors (or predecessors) of a regular set of configurations is itself regular. This is essential because even though the number of configuration can be infinite, we can still represent them symbolically using finite automata. In [14,4] efficient algorithms for computing both $Post^*$ and Pre^* were presented. They work by saturating an initial automaton $\mathcal{A}$ (representing some regular set of configurations) by adding new transitions and states to arrive at $\mathcal{A}_{post^*}$ (or $\mathcal{A}_{pre^*}$) automaton that represents the regular set of all its successor (or predecessor) configurations.

3.2 Representing processes and aspects

One of the challenges of using pushdown systems in our setting is the fact that there are no procedures and we do not have the concept of program locations. However, the behavior of aspects does remind of a stack. As an example, consider a process $A \,.\, B$, if the first action A is trapped by an aspect that gives advice $C \,.\, D \,.\, \textbf{proceed}$ then we suddenly have a process $C \,.\, D \,.\, A \,.\, B$ (where A should not be considered by this aspect again). But this can be thought of as a stack — we push two additional actions on it and then want to continue with the remaining ones. In other words we want the stack to characterize the process itself (i.e. the actions to be executed). The only remaining problem is how to ensure that the aspect will not trap action A again. Fortunately there is a solution for that — we can embed some additional information in the stack to indicate what aspect should consider the given action next. Apart from that, since the

stack is used for control flow, we can often improve the precision of our analyses by embedding in the stack some information about the communication that takes place. The most natural choices are the sender and receiver as well as the contents of the tuples that are sent or received. However, we usually do not want to consider all possible constants and all possible tuples that can arise at runtime — often just a subset of them along with some abstraction of the remaining ones will be enough. We will call them abstract constants and abstract tuples — since they abstract away from some of the possible runtime values. For instance in our example of an ATM and bank we are probably interested in when Ok and Abort can be sent and received, but the rest of the information (such as the amount of money to be withdrawn) is not that important, because it does not influence the control flow of the processes.

Therefore we define the stack alphabet Γ in the following way

$$\Gamma = ((\mathsf{Proc} \times \mathsf{Proc} \times \mathsf{Tuples}) \cup \mathsf{Internal}) \times (\mathsf{Asp} \cup \{\checkmark\})$$

where Proc is the set of processes, Asp is the set of aspects and ✓ is a special symbol indicating that all aspects have already analyzed the given action, Tuples is the set of all possible abstract tuples that are sent over the channels in the given system and Internal are internal actions of the process. Note that the definition of pushdown systems requires that these sets are finite. But this is not really a big restriction since we are already using abstract tuples and concrete processes are always finite. Furthermore, we need to define the set of actions. In our case this is actually quite similar to the stack elements. We define them as follows

$$\mathsf{Act} = (\mathsf{Proc} \times \mathsf{Proc} \times \mathsf{Tuples}) \cup \{\tau\}$$

The intuition here is that we do want to know the sender (first component of the tuple), the receiver (the second component) and what is communicated (the third component). This information will be essential in the subsequent section on communicating pushdown systems. Apart from that we also need to accommodate internal actions that are not important from the point of view of synchronization with other processes — thus the inclusion of τ that has the property that $a\tau = \tau a = a$.

Notice that in some cases we do not actually know what is sent/received in a given action (e.g. in **receive**$(!x)@\mathsf{N}$ we do not know what x might be). In such cases we can simply generate rules for all the possibilities. However, in many situations we could be quite a bit more clever about this — for instance it should be possible to generate such possibilities lazily, i.e. if some action is never pushed on a stack, we do not really need to add rules to pop it. Another example would be if we can determine that some constant is sent only between two processes, then we do not have to consider it when generating rules for actions of other processes.

Now let us get back to our example. Since we are only interested in Ok and Abort constants and the maximum arity of a tuple send by any process is equal to two, our set of abstract tuples will be:

$$\mathsf{Tuples} = \{(c) \mid c \in \mathsf{C}\} \cup \{(c_1, c_2) \mid c_1 \in \mathsf{C},\ c_2 \in \mathsf{C}\}$$

where $\mathsf{C} = \{\mathsf{Ok}, \mathsf{Abort}, *\}$ and $*$ stands for any constant other than Ok or Abort.

Rules for creating processes. To create a process the first thing that we do is to push all its actions on the stack. So if we have a process $\mathsf{P} :: a_1 \,.\, a_2 \,.\, a_3$ then we create a rule

$$\langle \mathsf{P},\ \Box \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{P},\ a_1\ a_2\ a_3 \rangle$$

where $\Box$ is a "start" symbol that can be used for creating the initial set of configurations (we will explain that later on). Furthermore this idea can be used to express recursion. Consider the following process: $\mathsf{Q} :: \mathbf{Rec}\mathsf{X} \,.\, a_1 \,.\, a_2 \,.\, \mathsf{X}$ By pushing X on the stack and treating it as a start symbol of the process we can easily model this recursion — the moment all actions are executed and X is on top of the stack, the rule is used to "recreate" the process:

$$\langle \mathsf{Q},\ \mathsf{X} \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{Q},\ a_1\ a_2\ \mathsf{X} \rangle$$

Apart from that we need to be able to handle choice. This can be achieved by creating all possible linear shapes of the process. For instance when generating the initial rules for process $\mathsf{P} :: a_1 \,.\, (a_2 + a_3)$ we would create:

$$\langle \mathsf{P},\ \Box \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{P},\ a_1\ a_2 \rangle \qquad \langle \mathsf{P},\ \Box \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{P},\ a_1\ a_3 \rangle$$

For the ATM in our example we can generate a set of rules for all choices of $x \in \mathsf{C}$ and $y \in \mathsf{C}$:

$$\left\{ \langle \mathsf{ATM},\ \Box \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{ATM},\ \begin{array}{l} (\mathsf{User}, \mathsf{ATM}, (x, y), \checkmark) \\ (\mathbf{check}(x), \mathsf{A}_1) \\ (\mathsf{ATM}, \mathsf{User}, (y), \checkmark) \\ (\mathsf{ATM}, \mathsf{Bank}, (x, y), \checkmark)) \end{array} \middle| \begin{array}{l} x \in \mathsf{C} \\ y \in \mathsf{C} \end{array} \right\}$$

As already mentioned, we can often be much smarter about generating the rules and create only a subset of the above (for instance Abort is never sent between User and Bank).

Rules for aspects. When generating the rules for the aspects we often have sufficient information in the stack element of the pointcut to be able to decide whether the aspect traps it or not. In the example above we could easily tell that some of the actions could never be trapped by any of our aspects (the aspects in the example trap only actions of the ATM). However, if we do not know whether the aspect will trap the action, we simply over-approximate and generate rules for both possibilities. One of the essential parts of generating the rules is to update the component of the stack that tracks what aspect should analyze the action next. In other words, if an action a is trapped by aspect A_1 then the **proceed** of the aspect should be the same action a but annotated with the next aspect. To make that clear, let us consider the internal **check** action of our ATM:

$$\left\{ \langle \mathsf{ATM},\ (\mathbf{check}, (x), \mathsf{A}_1) \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{ATM},\ \begin{array}{l} (\mathbf{check}, (x), \checkmark) \\ (\mathsf{ATM}, \mathsf{Bank}, (x), \checkmark) \\ (\mathsf{Bank}, \mathsf{ATM}, (y), \mathsf{A}_2) \rangle \end{array} \middle| \begin{array}{l} x \in \mathsf{C} \\ y \in \mathsf{C} \end{array} \right\}$$

As can be seen above, we have the internal action **check** with aspect A_1 on the left-hand side of the pushdown rule, but on the right-hand side we annotate it with ✓ as there are no more aspects that can match. This ensures that **check** will not be trapped by this aspect again.

Moreover, we must also handle the **if** conditions. Since we generate rules for various combinations of constants, we can often determine whether a condition is true at the stage of generating the rules. And if so, we can generate rules just for the right branch. However, in general this is not always possible. In such situations we can simply generate the rules for both cases, i.e. one if the condition is true and one if it is false. This corresponds to over-approximating the control flow. Therefore, from the point of view of precision, it might be beneficial to include in the set of abstract constants the ones that are used for comparisons.

Rules for executing actions. All of the above rules do not model the execution of any actions (and thus are considered as internal actions and labeled with τ). Execution in our context is nothing else than simply popping a stack element. So in general we simply create rules of the form $\langle p, a\rangle \overset{l}{\hookrightarrow} \langle p, \epsilon\rangle$ for all possible actions a that are annotated with ✓, where l is either τ if a is an internal action or a otherwise. An example from ATM is as follows:

$$\left\{ \langle \mathsf{ATM},\ (\mathsf{User}, \mathsf{ATM}, (x, y), \checkmark)\rangle \xhookrightarrow{(\mathsf{User},\mathsf{ATM},(x,y))} \langle \mathsf{ATM},\ \epsilon\rangle \middle| x \in \mathsf{C}\ y \in \mathsf{C} \right\}$$

which corresponds to execution of all actions where ATM receives a two-tuple from the user. Note that we require that the actions are annotated with ✓, which indicates that all aspects have been considered and the action can be executed.

Finally, we need to define the initial set of configurations whose successors we are interested in. This should clearly be the singleton with the name of the process (control location) and the start symbol (a single element stack). In case of ATM it is $\{\langle \mathsf{ATM}, \square\rangle\}$. Running the $Post^*$ algorithm on it will yield an automaton that represents all the possible future configurations of the thread. In this case it will describe what the process can be in the future (i.e. grow due to advice, shrink due to executing the actions, etc.).

3.3 Communicating pushdown systems

For now we have considered only a single thread at a time and we can create pushdown systems for each of them. However, since this construction does not take into account that communication takes place, it is quite an over-approximation of the behavior of the system. In this section we will remedy this and cover both the basic theory behind communicating pushdown systems, as well as how we can use them in our context.

Communicating pushdown systems have been introduced in [9,10] and subsequently used in [11]. We define them here with just slight modifications to accommodate for the fact that we handle message passing and not only synchronization as in [9,10].

Definition 2. *Communicating pushdown system (CPDS) is a tuple* $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ *of pushdown systems over the same set of actions* Act.

A global configuration of CPDS is a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots, \mathcal{P}_n$. We extend the relation $\stackrel{a}{\Longrightarrow}$ to global configurations in the following way:

- $g \stackrel{\tau}{\Longrightarrow} g'$ if there is $1 \leq i \leq n$ such that $c_i \stackrel{\tau}{\Longrightarrow} c_i'$ and $c_j' = c_j$ for all $j \neq i$
- $g \stackrel{(s,r,t)}{\Longrightarrow} g'$ if there are $i \neq j$ such that $c_i \stackrel{(s,r,t)}{\Longrightarrow} c_i'$ and $c_j \stackrel{(s,r,t)}{\Longrightarrow} c_j'$ ("s" stands for sender, "r" for receiver and "t" for tuple). Finally for all $k \neq i \wedge k \neq j$ we have that $c_k' = c_k$.

Using the pushdown systems we have the ability to characterize the set of all possible successors of some initial regular set of configurations. Moreover we want to annotate the result with the summarization of what happens on the paths to those successor configurations. Note that the pushdown rules are associated with actions, so each path in the transition system of a PDS has a corresponding sequence of actions, which is an element of the language generated by the set of actions. More formally it is a subset of the free monoid Act^* generated by the set of actions Act. Moreover, it is important to note that in general the paths of a process correspond to a context-free language [9,10].

Now let us consider a CPDS $(\mathcal{P}_1, \ldots \mathcal{P}_n)$ and assume that we are interested in the question whether a configuration from $C_1' \times \cdots \times C_n'$ is reachable from some configuration from $C_1 \times \cdots \times C_n$. In the following we will use $L_i = L(C_i, C_i')$ to denote the language summarizing all paths of the process i that go from any configuration of C_i to any configuration of C_i'. If we restrict ourselves, just for a moment, to the case where $n = 2$ then this problem is really nothing else than testing for emptiness of the intersection of the languages L_1 and L_2. The intuition behind this is that the intersection is empty only if there are no communication traces of the two processes that would match. However, the problem for reachability for arbitrary n is a bit more demanding. Consider a simple scenario with three process where process 1 first communicates with process 2, then with process 3 and then with 2 again. With the above "recipe" we could get that the $L_1 \cap L_2$ is empty. The problem is that the above does not account for the fact that process 1 communicates with process 3 and that this is not important from the point of view of synchronizing with process 2. Therefore in order to generalize this technique for arbitrary number of processes one has to accommodate for the interleaving of communication between different processes. Therefore we define $\widehat{L_i}$ to be an inverse homomorphic image of L_i

$$\widehat{L_i} = h_i^{-1} L_i$$

where h_i is defined as

$$h_i\ (s,r,t) = \begin{cases} (s,r,t) & \text{if } r = i \vee s = i \\ \epsilon & \text{otherwise} \end{cases}$$

The idea behind this is that process i allows for any communication that does not involve it, to take place between any of its actions. Intuitively when thinking about the pushdown system $\mathcal{P}_i$ we want to extend it with self-loops, which correspond to all possible communication actions of some other processes, on all its control locations. In [9,10] this problem is solved by introduction of interleaving (shuffle) operator. Also note that in case $n = 2$ we simply have $\widehat{L_i} = L_i$.

With the above we can finally reason about the reachability in a system of arbitrary many processes. More formally, if we have sets of global configurations $G = C_1 \times \ldots \times C_n$ and $G' = C'_1 \times \ldots \times C'_n$ and if

$$\widehat{L_1} \cap \ldots \cap \widehat{L_n} = \emptyset$$

then we can conclude that no configuration of G' is reachable from any configuration of G. However, there is still a minor problem with this approach — as already mentioned the generated languages are in general context-free and checking the emptiness of the intersection of context-free languages is undecidable. Therefore the next section will consider abstractions that can be used for over-approximation.

4 Analysis

4.1 Basic concepts

The *Post** and *Pre** algorithms can annotate the transitions of the $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ automata with "weights", i.e. some abstraction of the language generated by the set of actions. Our approach is based on [9,10] with some changes to accommodate for our slightly different context.

First of all we recall the definition of a semiring.

Definition 3. *A semiring is a tuple* $(D, \oplus, \otimes, \bar{0}, \bar{1})$ *such that*

- $(D, \oplus, \bar{0})$ *is a commutative monoid (hence* $\bar{0}$ *is a neutral element for* $\oplus$*)*
- $(D, \otimes, \bar{1})$ *is a monoid (hence* $\bar{1}$ *is a neutral element for* $\otimes$*)*
- $\otimes$ *distributes over* $\oplus$*, that is* $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *and* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
- $\bar{0}$ *is an annihilator for* $\otimes$*, that is* $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$

We consider an idempotent[3] semiring $(D, \oplus, \otimes, \bar{0}, \bar{1})$ with an associated abstract lattice $(D, \sqsubseteq, \sqcup, \bot)$ such that $a \sqsubseteq b$ iff $a \oplus b = b$, $\sqcup = \oplus$, $\bot = \bar{0}$. Furthermore we require that the lattice satisfies the ascending chain condition [15].

Moreover we also need to establish a *Galois connection* between the language generated by the transition system of the communicating pushdown system and our abstraction. Below we recall the definition of a Galois connection.

Definition 4. *A Galois connection is a tuple* (L, α, γ, M) *such that* L *and* M *are complete lattices and* α*,* γ *are monotone functions (called abstraction and concretization functions) that satisfy* $\alpha \circ \gamma \sqsubseteq \lambda m.m$ *and* $\gamma \circ \alpha \sqsupseteq \lambda l.l$*.*

[3] The operator $\oplus$ is additionally idempotent.

Intuitively a Galois connection specifies a semantically correct way to move our analysis from a precise lattice L (for which certain problems might be very hard or even undecidable) to a more abstract one M which has some desired computational properties. In our case we want to go from possibly infinite $\mathcal{P}(\mathsf{Act}^*)$ to one that allows $Post^*$ to terminate and gives us a decidable way of intersecting languages of various processes. We define the α and γ functions in the following way:

$$\alpha : \mathcal{P}(\mathsf{Act}^*) \to D$$
$$\alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \otimes \ldots \otimes v_{a_n}$$

$$\gamma : D \to \mathcal{P}(\mathsf{Act}^*)$$
$$\gamma(x) = \{a_1 \cdots a_n \mid v_{a_1} \otimes \ldots \otimes v_{a_n} \sqsubseteq x\}$$

where v_a is an abstract value of a (which will be defined by a particular abstraction). Note that $\alpha(\emptyset) = \bot$. Furthermore we also require that $\gamma(\bot) = \emptyset$. This gives us the desired property:

$$\forall L_1, \ldots, L_n : \alpha(L_1) \sqcap \ldots \sqcap \alpha(L_n) = \bot \implies L_1 \cap \ldots \cap L_n = \emptyset$$

Therefore, if these languages correspond to some paths between initial and target configurations, we know that there are no paths of those processes that are feasible when the communication is taken into account. This gives us the ability to prove that certain configurations of our system are not reachable.

Finally the only remaining thing to do is to actually compute the abstractions for each of the processes. Let C and C' be two regular sets of configurations and $\mathcal{A}_C$ and $\mathcal{A}_{C'}$ be the automata representing them. We consider the problem of computing $\alpha(L(C, C'))$. Assuming that we have used the $Post^*$ algorithm to compute the $\mathcal{A}_{post^*}$ weighted automaton. In the following we will use $\lambda(t)$ to denote the weight of the transition t. Since our automaton represents all the possible successors of C and we are only interested in some of them (only those in C'), we need to restrict the accepted configurations. In simple cases this can be achieved by querying the automaton for the weight of certain successors. But in general we can construct $\mathcal{A}^{C'}_{post^*}$ that is a restriction of $\mathcal{A}_{post^*}$ to the configurations in C'. To do that we can simply intersect $\mathcal{A}_{post^*}$ with $\mathcal{A}_{C'}$ automaton, that is create an automaton induced by the smallest set of transitions such that if $q_1 \xrightarrow{a} q_1'$ is in $\mathcal{A}_{post^*}$ and $q_2 \xrightarrow{a} q_2'$ is in $\mathcal{A}_{C'}$ then we have a transition $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ in $\mathcal{A}^{C'}_{post^*}$ and its weight is $\lambda(q_1 \xrightarrow{a} q_1')$. The result is exactly what we want — those successors of C that are in C'. And the weights represent the summarization of what happens along the paths between configurations in C and C'. The final step is to compute (using for instance a slightly modified algorithm presented in [7]):

$$\bigoplus \left\{ \lambda(w) \mid p \xrightarrow{w} q \in \mathcal{A}^{C'}_{post^*} \text{ where } p \text{ is an initial state and } q \text{ a final one} \right\}$$

where we extend λ to work over paths by using $\otimes$.

4.2 Abstraction

There are many possibile abstractions that can be used for these problems, for instance [9,10] presents a few options. Since in our problems we usually do not expect the stack to grow very large, we will use the i^{th}-prefix abstraction as introduced in [11]. The intuition behind it is that we simply impose a maximum length i that a word can have, i.e. we consider prefixes of words. The definition is as follows. Let W_i be the set of words of length less than ore equal to i. Then we define the semiring as follows: $D = \mathcal{P}(W_i)$, $\bar{0} = \emptyset$, $\bar{1} = \{\epsilon\}$, $\oplus = \cup$ and $U \otimes V = \{(uv)_i \mid u \in U,\ v \in V\}$ where $(w)_i$ is the prefix of w whose length is at most i. Moreover for every $a \in \mathsf{Act}$ we take $v_a = a$. Note that this automatically establishes a Galois connection where α and γ are defined as above.

4.3 Experiments

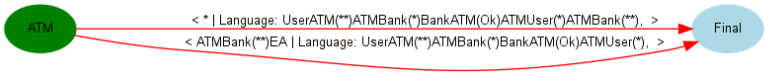

Fig. 1. Part of the simplified graph for the ATM.

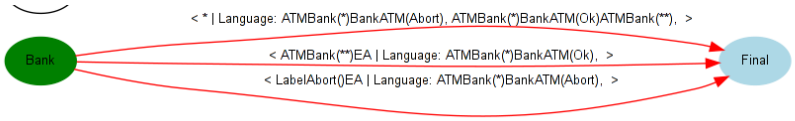

Fig. 2. Part of the simplified graph for the bank.

We have implemented this abstraction and used an off-the-shelf library WALi [16] for computing the $Post^*$ weighted automaton. Currently our implementation can be seen as a small library on top of WALi that offers higher level API capable of generating all pushdown rules (and their weights) for a given set of abstract constants. Since the resulting graphs are simply too large to include here, we have simplified the rules (without compromising the results) and present some of the more interesting parts in Fig. 1 and Fig. 2. The annotations on the edges are pairs of stack element (first component; `*` denotes an ϵ transition) and the weight of the transition (second component). Moreover `EA` stands for ✓ and `LabelAbort()` is an additional internal action that we inserted just after the action sending Abort to make it easier to see in the summarization of bank's actions at this point, which is:

`ATMBank(*)BankATM(Abort)`

Turning to the ATM, we can see that the process successfully dispenses the money and is about to inform the bank about the withdrawal with the following summarization of its communication:

```
UserATM(**)ATMBank(*)BankATM(Ok)ATMUser(*)
```

Now it should be clear that the intersection of the communication of the bank when it sends the abort message and the ATM when it dispenses the cash is empty — `BankATM(Ok)` and `BankATM(Abort)` do not match. This means that it is impossible for the bank and the ATM to reach this error configuration. In other words the cash will never be dispensed in a situation where the bank aborts the transaction.

5 Conclusions

In this paper we have considered a concurrent language equipped with message-passing primitives and support for the aspect-oriented paradigm. We believe that it also has a lot to offer in the context of coordination languages. In particular it gives us the ability to create very modular systems and separate unrelated functionality, which should make it easier to model complex systems.

However, the addition of aspects with advice allowing before or after actions leads to some interesting challenges. Those additional actions make it possible for a process to "grow" — one action trapped by an aspect can result in advice consisting of two or more actions. Furthermore, since the advice itself is analyzed by aspects, the processes can suddenly become arbitrarily large. Obviously this makes it much more difficult to analyze such systems. Our main contribution is to present an approach that is capable of solving analysis problems is such a context. To achieve this we used a technique from software model checking, namely communicating pushdown systems. Even though it is used mainly for analysis of recursive programs, we managed to adapt it to our setting. It proved to be a very useful and quite flexible tool, able to provide us with description of a process that can be arbitrarily large. Moreover, with the right abstraction, we can compute the summarization of its communication actions, allowing us to reason about the reachability in systems of concurrent threads. Since many safety problems can be reduced to reachability of error states, our approach can be used for verification purposes of such systems.

We believe that this approach can be adapted to various process calculi that use aspect-oriented paradigm. However, application to programming languages might pose additional challenges. Apart from already mentioned differences, we would also have to deal with two sources of recursion (procedures and aspects).

Finally, there are still some interesting future challenges. For instance, the question of how far can we extend the language and still be able to model it using pushdown systems. Moreover, from the point of view of efficiency we would prefer to generate only a small number of rules. On the other hand, to achieve better precision we would like to include as much information in the rules as possible. There is clearly a lot of room for experiments with various approaches and compromises depending on the situation.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. (1997) 220–242
2. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In Knudsen, J.L., ed.: ECOOP. Volume 2072 of Lecture Notes in Computer Science., Springer (2001) 327–353
3. Hankin, C., Nielson, F., Nielson, H.R., Yang, F.: Advice for coordination. In Lea, D., Zavattaro, G., eds.: COORDINATION. Volume 5052 of Lecture Notes in Computer Science., Springer (2008) 153–168
4. Schwoon, S.: Model-Checking Pushdown Systmes. PhD thesis, Technical University Munich (2002)
5. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural dataflow analysis. In Thomas, W., ed.: FoSSaCS. Volume 1578 of Lecture Notes in Computer Science., Springer (1999) 14–30
6. Reps, T.W., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In Cousot, R., ed.: SAS. Volume 2694 of Lecture Notes in Computer Science., Springer (2003) 189–213
7. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. **58**(1-2) (2005) 206–263
8. Yang, F.: Aspects with Program Analysis for Security Policies. PhD thesis, Technical University of Denmark (2010)
9. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003) 62–73
10. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. **14**(4) (2003) 551–
11. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing c programs with recursive calls. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 334–349
12. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In Ramakrishnan, C.R., Rehof, J., eds.: TACAS. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 282–298
13. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8) (1978) 666–677
14. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: CAV. (2000) 232–247
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis (2. corr. print). Springer (2005)
16. Kidd, N., Lal, A., Reps, T.W.: Wali: The weighted automaton library (December 2007)