Weighted Dynamic Pushdown Networks

Alexander Wenner

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
alexander.wenner@uni-muenster.de

Abstract. We develop a generic framework for the analysis of programs with recursive procedures and dynamic process creation. To this end we combine the approach of weighted pushdown systems (WPDS) with the model of dynamic pushdown networks (DPN). The resulting model, weighted dynamic pushdown networks (WDPN), describes processes running in parallel, each of them being able to perform pushdown actions, that may spawn new processes as a side effect. As with WPDS, transitions are labelled by weights to carry additional information. Starting from techniques for WPDS and DPN, we derive a method to determine meet-over-all-paths values for the paths between regular sets of configurations of a WDPN. Using this method we are able to solve basic dataflow analysis problems in a parallel context.

1 Introduction

The interest in writing parallel programs has increased in recent years. However parallel programming is notoriously difficult and error-prone. Thus static analysis of parallel programs has become more and more important. The goal of this paper is to present a generic framework for the analysis of parallel programs, especially in the presence of recursive procedures and dynamic process creation. We base our framework on DPN [1] and WPDS [2]. DPN precisely model procedures and process creation and have been studied for reachability analyses. Since the analysis of recursive procedures and synchronisation is undecidable [3], DPNs do not model synchronisation between processes. However, through the addition of weights we will be able to analyse some interaction between processes. WPDS extend pushdown systems (PDS) by labelling transitions with weights and solving the generalised pushdown predecessor (GPP) problem, which is the meet-over-all-paths solution for paths from a starting configuration into a regular set of target configurations. The weights can be used to formulate a wide range of analysis problems. The GPP problem formulation allows for a specific query represented by a regular constraint on the shape of the call-stack, in contrast to standard dataflow techniques, where typically all information at the topmost program point is merged.

The main advantage of our framework is, that we extend this ability to formulate a query depending on a regular constraint on the shape of the call-stack to queries depending on a regular constraint on the shape of the entire network.

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 590–609, 2010.

```
main:                  init_worker:          worker:
1: call init_worker    5: spawn worker       7: write a
2: write a             6: return             8: return
3: use a
4: return
```

Fig. 1. Example program

Consider the pseudo program in Figure 1. It calls a procedure to initialise a `worker` process, that calculates a value which is then stored in the variable `a`. In parallel the `main` process uses the variable to store a value it needs in a following step. The program obviously contains a data race, since the `worker` can overwrite the value of `a` before the `main` process reads it. Our framework is now able to refine the analysis of such a data race by distinguishing the situation where the `main` process reaches the use and the `worker` process has completed his computation from the situation where the `worker` process has completed no or only some steps.

Up to this point our framework can solve the bitvector problems for DPNs formulated in [1], which is able to handle the same refinement described above. The automata based approach in [1] however requires multiple computations of predecessor sets, whereas our method only needs one step. The shortest path analysis from [2] is an example for an analysis with an infinite domain, which can not be formulated using the automata based techniques from [1], but can be easily handled by our framework. In [4] a different approach to generalize WPDS to parallel programs is presented, by introducing a context bound. This approach can handle more powerful analyses than our framework, but the introduction of a context bound leads to an underapproximation, whereas our approach handles unbounded context switches precisely.

A main result is, that our framework can handle all KILL/GEN analyses precisely in a uniform way. To the best of our knowledge no more general class of analyses is known, for which precise analysis for some class of parallel program is possible. In [5,6,7] KILL/GEN analyses have been considered for pcall type parallelism, which can not be used to accurately model the process creation of a DPN [1], which is the basis of our framework. In [8] KILL/GEN analyses were extended to a model similar to DPN, which can handle dynamic process creation. However in this approach all dataflow information reaching a program point is merged, regardless of the state of the rest of the network. As described above, our framework allows for a more distinct query, depending on the state of the whole network.

Approach. Analogous to WPDS we extend DPN to WDPN by annotating weights to transitions and study the GPP problem. Even though a WPDS is then simply a WDPN with one process, adapting the approach to solve the GPP problem from WPDS to WDPN is problematic. In general a path of a DPN is an interleaving of the transitions of arbitrary many parallel processes. Results from [1] show, that the set of paths connecting two regular sets of configurations can not be described

in a way, where standard techniques like abstract interpretation [9] can be applied to compute the abstraction in the weight domain.

We avoid these problems by introducing a branching semantics for DPN similar to the tree semantics in [10]. Transitions of newly spawned processes are no longer mixed with the transitions of the creating process, but contained in their own branch. This results in executions which are tree shaped for single processes and form hedges, which contain a tree for each process, for configurations with multiple processes. The set of hedges connecting two regular sets of configurations can be described by a constraint system, adapting the approach for WPDS.

We introduce a weight domain to abstract these trees, and study the analogous branching GPP (BGPP) problem, which is the meet-over-all-hedges solution, for these branching WDPN (BWDPN). The solution of the BGPP problem can be obtained by abstract interpretation of the constraint system. We show, that if the weight domain of a WDPN and the extended weight domain of a BWDPN, based on the same DPN, are related, the solution for the GPP problem of the WDPN can be derived from the solution of the corresponding BGPP problem of the BWDPN.

We demonstrate how this framework of WDPN and BWDPN can be used to solve shortest path problems, bitvector analyses and the more general KILL/GEN analyses for programs with recursive procedures and dynamic thread creation.

Outline. The remainder of the paper is organised as follows: Section 2 presents the intuitive extension of WPDS to DPN called WDPN and defines the GPP. Section 3 introduces BWDPN. We formulate the BGPP problem and present the relation to the GPP problem. Section 4 presents applications and Section 5 introduces the approach to solve the BGPP problem for BWDPN.

2 Weighted Dynamic Pushdown Networks

A DPN [1] is a model for parallel programs with multiple processes and dynamic process creation. Each process is modeled as a PDS, where the rules are extended to allow creation of new processes. Formally a DPN is a tuple $\mathcal{M} = (P, \Gamma, \Delta)$, where P is a finite set of control states and Γ is a finite set of stack symbols, with $P \cap \Gamma = \emptyset$. Δ is a finite set of transition rules of the form:

$$p\gamma \hookrightarrow c \text{ with } p \in P, \gamma \in \Gamma, c \in (P\Gamma^*)^* P\Gamma^*.$$

The right side of a rule consists of the new control state and stacktop of the original process in the rightmost position and the control states and stacks of all processes spawned by this rule to the left. Configurations of a DPN are words from $\mathsf{Conf} = (P\Gamma^*)^*$. The empty configuration is written as ε. For the rest of the paper we fix a DPN $\mathcal{M} = (P, \Gamma, \Delta)$ and two regular sets $C_1, C_2 \subseteq \mathsf{Conf}$.

Example 1. The program in Figure 1 leads to a DPN with rules $r_1 = p\gamma_1 \hookrightarrow p\gamma_5\gamma_2$, $r_2 = p\gamma_5 \hookrightarrow p\gamma_7 p\gamma_6$, $r_3 = p\gamma_6 \hookrightarrow p$, $r_4 = p\gamma_2 \hookrightarrow p\gamma_3$, $r_5 = p\gamma_3 \hookrightarrow p\gamma_4$

and $r_6 = p\gamma_7 \hookrightarrow p\gamma_8$, where a stack symbol γ_i represents the control location at the beginning of line i of the program. The set of starting configurations for the analyses of our program described in the introduction would be $C_1 = \{p\gamma_1\}$ and the target sets would either be $C_2 = \{p\gamma_8 p\gamma_3\}$ if the spawned process makes all steps or $C_2' = \{p\gamma_7 p\gamma_3\}$ if it makes no steps.

Interleaving Semantics. An execution of the DPN $\mathcal{M}$ is represented by a path. A path is defined as a sequence of rules:

$$\rho = r_1 \dots r_n \text{ with } r_i \in \Delta.$$

The empty path is denoted by ε_ρ and Paths is the set of all paths. The execution of a path is modeled by the labelled transition relation $\longrightarrow \subseteq \mathsf{Conf} \times \mathsf{Paths} \times \mathsf{Conf}$, similar to [1], with:

$$[\mathsf{empty}]\ c \xrightarrow{\varepsilon_\rho} c \quad [\mathsf{rule}]\ up\gamma v \xrightarrow{r\rho} c \text{ if } r = p\gamma \hookrightarrow c', uc'v \xrightarrow{\rho} c$$

Application of a rule replaces the control state and top symbol of one stack by the new control state and stacktop specified by the rule and inserts the newly created processes with their initial stacks, as defined by the rule, to the left. We call this the interleaving semantics of the DPN, since the rules of all processes are mixed up. We are interested in the set:

$$\mathsf{Paths}(C_1, C_2) = \{\rho \in \mathsf{Paths} \mid \exists c_1 \in C_1, c_2 \in C_2 \text{ with } c_1 \xrightarrow{\rho} c_2\},$$

of connecting paths from C_1 to C_2.

Example 2. The sets of connecting paths in our example are $\mathsf{Paths}(C_1, C_2) = \{r_1r_2r_6r_3r_4, r_1r_2r_3r_6r_4, r_1r_2r_3r_4r_6\}$ and $\mathsf{Paths}(C_1, C_2') = \{r_1r_2r_3r_4\}$.

Weights. In order to abstract from the set of connecting paths to the aspects which are relevant to the analysis, we assign a weight to each transition of the DPN. The structure of the weight domain is captured by a complete idempotent semiring, which supports the necessary operators $\odot$ for concatenation of weights along a path and $\oplus$ for combination of weights of different paths. A complete idempotent semiring is a tuple $\mathcal{S} = (D, \oplus, \odot, 0, 1)$, where D is a set of elements with $0, 1 \in D$ and $\oplus, \odot$ are binary operators on D with:

- $(D, \oplus)$ is a commutative monoid with neutral element 0 and $\oplus$ is idempotent
- $(D, \odot)$ is a monoid with neutral element 1 and 0 annihilates $\odot$
- $(D, \sqsubseteq)$ is a complete lattice, where $\sqsubseteq$, with $d_1 \sqsubseteq d_2 :\Leftrightarrow d_1 \oplus d_2 = d_1$ for $d_1, d_2 \in D$, is the partial order induced by $\oplus$, i.e. $\oplus$ is the meet operator of the lattice $(D, \sqsubseteq)$ and 0 is the $\top$-element
- $\odot$ distributes over arbitrary $\oplus$, i.e. $\bigoplus D_1 \odot \bigoplus D_2 = \bigoplus\{d_1 \odot d_2 \mid d_i \in D_i\}$ for $D_1, D_2 \subseteq D$

We fix a semiring $\mathcal{S} = (D, \oplus, \odot, 0, 1)$. The weights are assigned to the transitions of the DPN $\mathcal{M}$ using a weight function $f : \Delta \to D$. The function depends on the

current analysis, since it describes how the transitions of the DPN are connected to the analysed information represented by the semiring. We assume a given weight function f for the rest of the paper. The tuple $\mathcal{W} = (\mathcal{M}, \mathcal{S}, f)$ is called a WDPN. Given the WDPN we define an abstraction function $\alpha : \mathsf{Paths} \to D$ for paths:

$$[\mathsf{empty}]\ \alpha(\varepsilon_\rho) = 1 \quad [\mathsf{rule}]\ \alpha(r\rho) = f(r) \odot \alpha(\rho)$$

Overloading it for sets of paths with $\alpha(M) = \bigoplus\{\alpha(\rho) \mid \rho \in M\}$, we can formulate the GPP problem for WDPN as computing:

$$\delta(C_1, C_2) = \alpha(\mathsf{Paths}(C_1, C_2)).$$

3 Branching Weighted Dynamic Pushdown Networks

It follows from results in [1] that the set $\mathsf{Paths}(C_1, C_2)$ can not be characterised as least solution of a constraint system which uses only operators to concatenate or interleave paths. Therefore we can not compute the solution for the GPP problem directly by an abstract interpretation [9] of such a constraint system. To avoid this problem we consider an alternative interpretation of an execution of a DPN in form of a tree or hedge, first introduced in [10]. We will later see, that set of connecting hedges can be assembled from sets of partial trees, which in turn can be characterised using a constraint system.

Branching Semantics. We recursively define the set Trees of execution trees, where $\mathsf{Hedges} = \mathsf{Trees}^*$ is the set of execution hedges.

$$\varepsilon_\tau \in \mathsf{Trees} \quad r(\sigma\tau) \in \mathsf{Trees} \ \text{ for } r \in \Delta, \sigma \in \mathsf{Hedges}, \tau \in \mathsf{Trees}$$

The empty tree ε_τ consisting of a single leaf node, representing a finished execution, is a tree. $r(\sigma\tau)$ is a tree with a root node labelled with a rule $r \in \Delta$, describing the first step of the execution, and an ordered list of subtrees $\sigma\tau \in \mathsf{Hedges}$, representing the executions σ of spawned processes and the rest of the execution τ of the spawning process. The order of the children corresponds to the order of processes on the right side of the rule r. ε_σ is the empty hedge.

The execution of a hedge is modeled by the labelled transition relation $\Longrightarrow \subseteq \mathsf{Conf} \times \mathsf{Hedges} \times \mathsf{Conf}$, with:

$$[\mathsf{none}]\ \ \varepsilon \overset{\varepsilon_\sigma}{\Longrightarrow} \varepsilon \qquad [\mathsf{tree}]\ cpw \overset{\sigma\tau}{\Longrightarrow} c'c'' \ \text{ if } c \overset{\sigma}{\Longrightarrow} c' \text{ and } pw \overset{\tau}{\Longrightarrow} c''$$
$$[\mathsf{empty}]\ pw \overset{\varepsilon_\tau}{\Longrightarrow} pw \quad [\mathsf{rule}]\ p\gamma w \overset{r(\sigma)}{\Longrightarrow} c \qquad \text{if } r = p\gamma \hookrightarrow c', c'w \overset{\sigma}{\Longrightarrow} c$$

We call this the branching semantics of the DPN, since each process has its own branch in the execution. We are interested in the set

$$\mathsf{Hedges}(C_1, C_2) = \{\sigma \in \mathsf{Hedges} \mid \exists c_1 \in C_1, c_2 \in C_2 \text{ with } c_1 \overset{\sigma}{\Longrightarrow} c_2\},$$

of connecting hedges.

Example 3. The sets of connecting hedges for our example are $\mathsf{Hedges}(C_1, C_2) = \{r_1(r_2(r_6(\varepsilon_\tau)r_3(r_4(\varepsilon_\tau))))\}$ and $\mathsf{Hedges}(C_1, C_2') = \{r_1(r_2(\varepsilon_\tau r_3(r_4(\varepsilon_\tau))))\}$.

We define the **;** operator to concatenate a hedge to the last tree of a hedge:

$$[\mathsf{hedge}]\ (\sigma\tau)\,;\sigma' = \sigma(\tau\,;\sigma') \quad [\mathsf{empty}]\ \varepsilon_\tau\,;\sigma' = \sigma' \quad [\mathsf{rule}]\ r(\sigma)\,;\sigma' = r(\sigma\,;\sigma')$$

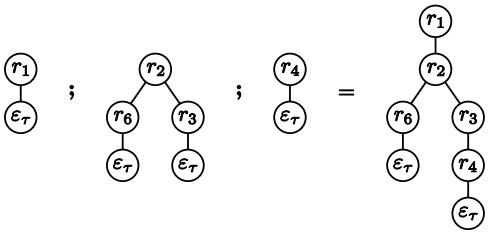

Fig. 2. Example for the concatenation of trees

Appending a hedge removes the rightmost leaf of the first hedge and adds the trees of the second hedge as new children. Thus, if you only consider concatenation of trees, it is simply concatenation along the rightmost branches. The reason for defining concatenation this way is, that we will later see, that we can assemble any execution tree for an initial process by concatenating trees from a finite number of classes. In the context of program analysis, these classes represent executions inside the body of a procedure. Figure 2 shows how we can assemble an execution of our example program by concatenating the call rule r_1, the execution inside the called `init_worker` procedure, containing the execution of the spawned process, and the rest of the execution of the `main` procedure. We extend concatenation of trees to concatenation of hedges to describe the construction of a new tree from a rule and a list of subtrees as concatenation of the tree with the rule as root node and the empty tree as only child and the hedge formed by the list of subtrees.

Interleaving vs. Branching. There is a strong connection between the interleaving and branching semantics of a DPN. A hedge represents of a set of paths, which can be constructed by interleaving the branches and trees of the hedge. Consider a function $\psi : 2^{\mathsf{Hedges}} \to 2^{\mathsf{Paths}}$ that computes the set of interleavings of a set of hedges, here $\|$ is used for the standard interleaving operator for paths:

$$\begin{array}{ll} [\mathsf{none}]\ \psi(\varepsilon_\sigma) = \{\varepsilon_\rho\} & [\mathsf{tree}]\ \psi(\sigma\tau) = \psi(\sigma) \parallel \psi(\tau) \\ [\mathsf{empty}]\ \psi(\varepsilon_\tau) = \{\varepsilon_\rho\} & [\mathsf{rule}]\ \psi(r(\sigma)) = r\psi(\sigma) \end{array}$$

Results from [10] show, that:

Theorem 4. *We have:*

$$\mathsf{Paths}(C_1, C_2) = \psi(\mathsf{Hedges}(C_1, C_2)).$$

Extended Weights. The semiring structure used for WDPN is not suitable to abstract hedges. Especially with regard to the approach of combining an execution tree out of partial trees by concatenation. The semiring could be used to compute a weight for a given tree by computing the meet-over-all-interleavings, using the weights given by f for each rule. However in this case the operator $\odot$ is not useable as abstraction for concatenation of trees, since the interleaving of a concatenated tree would in general not be the same as the concatenation of the interleavings of the partial trees. If we take the trees from Figure 2 and set $f(r_i) = w_i$, the left side evaluates to $w_1 \odot w_2 \odot (w_6 \odot w_3 \oplus w_3 \odot w_6) \odot w_4$, which in general is different from the value $w_1 \odot w_2 \odot (w_6 \odot w_3 \odot w_4 \oplus w_3 \odot w_6 \odot w_4 \oplus w_3 \odot w_4 \odot w_6)$ of the right hand side.

To abstract hedges we define an extended complete idempotent semiring, which contains the additional $\bar{\otimes}$ operator for parallel combination of weights. By making the parallel composition explicit and introducing new weights, we can store additional information in the weights concerning parallel branches to delay the actual interleaving. An extended complete idempotent semiring $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, \bar{0}, \bar{1})$ is a tuple, where E is a set of values and $\bar{\oplus}, \bar{\odot}, \bar{\otimes}$ are binary operators on E with:

- $(E, \bar{\oplus}, \bar{\odot}, \bar{0}, \bar{1})$ is a complete idempotent semiring
- $(E, \bar{\otimes})$ is a semigroup, $\bar{1} \bar{\otimes} e = e$ for $e \in E$ and $\bar{0}$ annihilates $\bar{\otimes}$
- $\bar{\otimes}$ distributes over arbitrary $\bar{\oplus}$, i.e. $\bigbar{\oplus} E_1 \bar{\otimes} \bigbar{\oplus} E_2 = \bigbar{\oplus}\{e_1 \bar{\otimes} e_2 \mid e_i \in E_i\}$ for $E_1, E_2 \subseteq E$
- $(e_1 \bar{\otimes} e_2) \bar{\odot} e_3 = e_1 \bar{\otimes} (e_2 \bar{\odot} e_3)$, for $e_1, e_2, e_3 \in E$

The fourth property ensures, that **;** is abstracted by $\bar{\odot}$, by always appending weights to the rightmost weight of a parallel combination. In this regard the $\bar{\otimes}$ operator differs from the abstract interleaving operator $\otimes$ introduced in [6]. The new operator is especially not commutative. This can also be seen in the fact, that $\bar{1}$ is only left identity for $\bar{\otimes}$, since a $\bar{1}$ in the right component can be altered by appending an additional weight.

We fix an extended semiring $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, \bar{0}, \bar{1})$. As with WDPN we assume, that a weight function $\bar{f} : \Delta \to E$ is given. The tuple $\mathcal{B} = (\mathcal{M}, \mathcal{E}, \bar{f})$ is called a BWDPN. Given a BWDPN we define an abstraction function $\beta : \mathsf{Hedges} \to E$ for hedges:

$$\begin{array}{llll} [\mathsf{none}] & \beta(\varepsilon_\sigma) = \bar{1} & [\mathsf{tree}]\ \beta(\sigma\tau) & = \beta(\sigma) \bar{\otimes} \beta(\tau) \\ [\mathsf{empty}]\ \beta(\varepsilon_\tau) = \bar{1} & & [\mathsf{rule}]\ \beta(r(\sigma)) & = \bar{f}(r) \bar{\odot} \beta(\sigma) \end{array}$$

Overloading it for sets of hedges with $\beta(M) = \bigbar{\oplus}\{\beta(\sigma) \mid \sigma \in M\}$, we define the BGPP problem for BWDPN as computing:

$$\theta(C_1, C_2) = \beta(\mathsf{Hedges}(C_1, C_2)).$$

Weights vs. Extended Weights. At this point, we have formulated two problems. The GPP problem describes the meet-over-all-paths of the interleaving semantics, the BGPP problem describes the meet-over-all-hedges of the branching semantics. As mentioned in the beginning of this section, the solution to

the GPP problem can not be computed directly. However we will later see, that the solution of the BGPP problem can be obtained by solving a constraint system. In the previous paragraph, we have seen, that we can not simply use the weight domain for the GPP problem as a weight domain for the corresponding BGPP problem. However Theorem 4 describes a strong relation between the set of reaching paths and the set of reaching hedges. A similar result can be shown for the solutions of the GPP and BGPP problems, if the semiring of the WDPN is related to the extended semiring of the BWDPN. We describe the necessary relation by an extension. An extension is a tuple $(\mathcal{S}, \mathcal{E}, \iota, \eta)$, containing embedding and projection functions $\iota : D \to E$ and $\eta : E \to D$, where for $d, d_i \in D, e, e_i \in E$ the following conditions hold:

- E is the smallest set with $\iota(D) \subseteq E$, closed under $\bar{\odot}, \bar{\otimes}$ and arbitrary $\bar{\oplus}$
- $\iota(0) = \bar{0}$, $\iota(1) = \bar{1}$ and $\eta(\iota(d)) = d$
- η distributes over arbitrary $\bar{\oplus}$, i.e. $\eta(\bigoplus M) = \bigoplus\{\eta(e) \mid e \in M\}$ for $M \subseteq E$
- $\eta(e \mathbin{\bar{\otimes}} \bar{1}) = \eta(e)$
- $\eta(e_1 \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e_n) = \bigoplus_{i=1}^{n} d_i \odot \eta(e_1 \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e'_i \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e_n)$ for $e_i = \iota(d_i) \mathbin{\bar{\odot}} e'_i$

The first three points ensure, that every weight of the original semiring has a corresponding weight in the extended semiring and in reverse every element of the extended semiring is a combination of embedded weights of the original semiring. The last two points ensure, that the combination of weights is mapped to the meet-over-all-interleavings of the weights they are constructed from. For the rest of the paper, we assume that the semiring $\mathcal{S}$ and the extended semiring $\mathcal{E}$ are connected by the extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$.

If $\bar{f}(r) = \iota(f(r))$, for all $r \in \Delta$, i.e. the analysis of the WDPN is embedded in the BWDPN, we can prove $\alpha(\psi(\sigma)) = \eta(\beta(\sigma))$ for all $\sigma \in \mathsf{Hedges}$ by induction on σ. Consequently with Theorem 4:

Theorem 5. *It follows, that:*

$$\delta(C_1, C_2) = \eta(\theta(C_1, C_2)).$$

Construction of Extended Semiring and Extension. An example for an extended semiring $\mathcal{E}$ and extension $(\mathcal{E}, \mathcal{S}, \iota, \eta)$, which exists for any semiring $\mathcal{S}$, is the extended semiring of weighted hedges, i.e. hedges where nodes are labelled with a weight from the semiring $\mathcal{S}$. This abstraction contains nearly all information contained in the execution trees. An abstraction of an execution hedge is simply the hedge, where nodes previously annotated with r are now annotated with $f(r)$ and the empty tree is annotated with 1. Interior nodes labelled with the neutral element 1, which have no influence on the total weight of the tree, are removed.

We define the set of weighted trees WTrees recursively, where $\mathsf{WHedges} = \mathsf{WTrees}^+$ is the set of weighted hedges:

$$1 \in \mathsf{WTrees} \quad w(\sigma) \in \mathsf{WTrees} \text{ for } w \in D \setminus \{1\}, \sigma \in \mathsf{WHedges}$$

1 is the empty weighted tree consisting of a single leaf node labelled with 1 and $w(\sigma)$ is a weighted tree with a root node labelled with w and children σ. 1 doubles as the initial empty weighted hedge, and we define $1\sigma = \sigma$.

The operations of the extended semiring are then mapped onto the corresponding weighted tree operations. We define $\sigma \bar{\otimes} \sigma' = \sigma\sigma'$ and concatenation is concatenation of weighted hedges as with execution hedges:

$$[\mathsf{hedge}]\ \sigma\tau \,\bar{\odot}\, \sigma' = \sigma(\tau \,\bar{\odot}\, \sigma') \quad [\mathsf{empty}]\ 1 \,\bar{\odot}\, \sigma = \sigma \quad [\mathsf{rule}]\ w(\sigma) \,\bar{\odot}\, \sigma' = w(\sigma \,\bar{\odot}\, \sigma')$$

Since there is no obvious way to compute a meet for two weighted hedges, we go to the powerset of $\mathsf{WHedges}$. The extended semiring is given by $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, \emptyset, \{1\})$, where $E = 2^{\mathsf{WHedges}}$, $\bar{\oplus} = \cup$ and $\bar{\odot}, \bar{\otimes}$ are extended to sets.

The embedding of the corresponding extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$, transforms a weight into a corresponding set of weighted trees:

$$\iota(0) = \emptyset \quad \iota(1) = \{1\} \quad \iota(w) = \{w(1)\}$$

The projection back into the semiring then computes the value of all interleavings for a given weighted hedge:

$$\eta(1) = 1 \quad \eta(\sigma 1) = \eta(\sigma)$$
$$\eta(w_1(\sigma_1) \dots w_n(\sigma_n)) = \bigoplus_{i=1}^{n} w_i \odot \eta(w_1(\sigma_1) \dots \sigma_i \dots w_n(\sigma_n))$$

η is extended to sets by $\eta(M) = \bigoplus\{\eta(\sigma) \mid \sigma \in M\}$ for $M \subseteq \mathsf{WHedges}$. It can be easily seen, that the definitions fulfill all the conditions for an extension between the semiring $\mathcal{S}$ and $\mathcal{E}$.

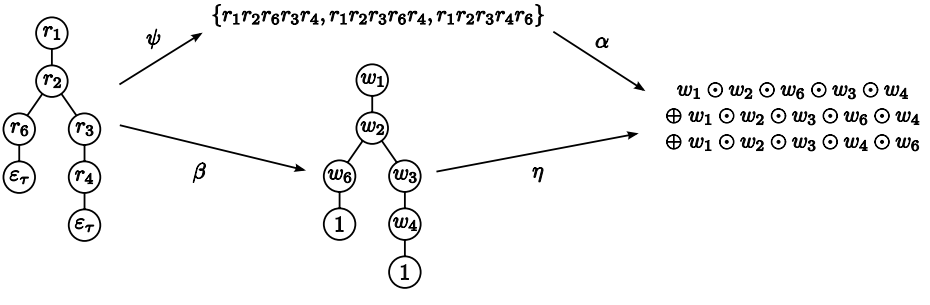

Fig. 3. Different abstractions provide the same result

Figure 3 shows, that in our example from Section 1, the abstraction, with $f(r_i) = w_i$, of the set of paths of the connecting hedges $\mathsf{Hedges}(C_1, C_2)$ and the projection of the direct abstraction, with $\bar{f}(r_i) = \iota(w_i)$, of the same hedges lead to the same results, confirming the result of Theorem 5.

However, since the size of the sets, trees and hedges is not bounded, this extended semiring is not efficient. In the next section we will explain, how in some cases a smaller representation for the weighted hedges can be found, that can be used to compute a solution for the BGPP problem.

4 Applications

Since the existence of an efficient extended semiring and a matching extension for a given semiring is not self-evident, we first give some examples of semirings, for which an efficient extended semiring and a corresponding extension can be constructed, before describing the approach to solve the BGPP problem in Section 5.

Starting form the weighted trees in the previous section, we can simplify the appearance of a tree by collapsing sequential parts of a tree using the $\odot$ operator of the semiring. To still be able to compute an interleaving of two collapsed branches, we assume an abstract interleaving operator $\otimes$. The existence of an abstract interleaving operator is again not self-evident, but is given for the applications described later in this section. Since a weighted tree can be a representation of a partial execution, we can not yet interleave the rightmost branch with any of the other branches, since it only represents part of the execution of the rightmost process. The solution to this problem is to precompute the total weight of the tree for all possible weights of appended weighted trees. Thus the extended weight representing a collapsed weighted tree is a function from D to D. Figure 4 visualizes the collapsing of the weighted hedge in our example into a function.

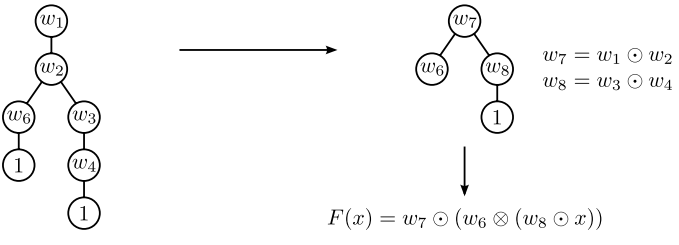

Fig. 4. Collapsing of a weighted tree

We set $E \subseteq \{F : D \to D\}$. The embedding of a weight $d \in D$ represents a tree with just a single node labelled by the embedded weight. Concatenation of another weighted tree leads to concatenation of d with the weight of the tree, hence $\iota(d) = F_d$, with $F_d(x) = d \odot x$. The projection is then a simple evaluation of the collapsed tree, where the empty weighted tree, with semiring weight 1 is appended, hence $\eta(F) = F(1)$. Concatenation of collapsed trees is then combination of the functions representing the trees $(F \bar{\odot} G)(x) = F(G(x))$ and the meet is the pointwise meet $(F \bar{\oplus} G)(x) = F(x) \oplus G(x)$. For interleaving, the left branch is evaluated at 1 to get the total weight of the left tree and the interleaving with all possible values for the right tree is precomputed, hence $(F \bar{\otimes} G)(x) = F(1) \otimes G(x)$. Then E is the smallest set with $\iota(D) \subseteq E$, which is closed under $\bar{\odot}, \bar{\otimes}$ and arbitrary $\bar{\oplus}$.

Shortest Path Analysis. The shortest path analysis assigns a positive integer weight to all transitions. The weight of a path is the sum of the weights of the transitions occurring on the path. The goal is to find the weight of the path with the smallest weight. We use the semiring $\mathcal{S} = (\mathbb{N} \cup \{\infty\}, min, +, \infty, 0)$ introduced in [2]. Since $+$ is commutative and associative, the order in which transitions occur and are combined on a path is irrelevant. Thus $+$ can be used as the abstract interleaving operator $\otimes$.

If we apply the construction described above, we get an extended semiring $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, F_\infty, F_0)$, with $\bar{\oplus}, \bar{\odot}$ and $\bar{\otimes}$ as described. Furthermore, we get an extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$, with $\iota(d)(x) = d + x$, for $d \in \mathbb{N} \cup \{\infty\}$ and η as described. Since E contains only the elements derived from elements in $\iota(D)$, it can be show that all elements $F \in E$ can be written as $F(x) = d_F + x$ with $d_F \in \mathbb{N} \cup \{\infty\}$. Then the operators of the extended semiring and extension can be reduced to the operators of the semiring as follows:

$$\begin{array}{llll} (F \mathbin{\bar{\odot}} G)(x) = (d_F + d_G) + x & & (F \mathbin{\bar{\oplus}} G)(x) = min\{d_F, d_G\} + x \\ (F \mathbin{\bar{\otimes}} G)(x) = (d_F + d_G) + x & & \eta(F) = d_F \end{array}$$

One can observe, that instead of the functional notation of $\mathcal{E}$, one can simply use $(\mathbb{N} \cup \{\infty\}, min, +, +, \infty, 0)$ as extended semiring. Using this construction, we can precisely compute the length of the shortest path connecting the starting and the target set in the DPN.

Example 6. By setting $f(r_4) = f(r_6) = 1$ and $f(r) = 0$ for all other rules, one can, for example, determine the minimum number of times a is written in our example.

Bitvector Analyses. Bitvector analyses examine a property represented by a single bit. For lack of space, we consider only forward, information is propagated from the start of the program, must, all paths reaching a target configuration must set the bit to 1, bitvector analyses. Backward, information is propagated from the end of the program, or may, it suffices, that one path reaching a target configuration sets the bit to 1, analyses can be handled similarly. The transitions of the DPN are annotated with transformers, that change the current state of the bit. We use the semiring $\mathcal{S} = (D, \oplus, \odot, \mathsf{zero}, \mathsf{id})$, where $D = \{\mathsf{kill}, \mathsf{id}, \mathsf{gen}, \mathsf{zero}\}$. Here gen represents the transformer setting the bit to 1, id is the identity and kill sets the bit to 0. The artificial weight zero is introduced to represent the zero element of the ring. For a forward analysis, $\odot$ is reversed functional combination extended to include zero. In case of a must analysis $\oplus$ is a meet operator inducing the ordering $\mathsf{kill} \sqsubseteq \mathsf{id} \sqsubseteq \mathsf{gen} \sqsubseteq \mathsf{zero}$. In [6] it was shown, that the operator $\otimes$, defined as $f \otimes g = (f \odot g) \oplus (g \odot f)$, is an abstract interleaving operator on the path level.

If we apply the construction described above, we get an extended semiring $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, F_{\mathsf{zero}}, F_{\mathsf{id}})$, with $\bar{\oplus}, \bar{\odot}$ and $\bar{\otimes}$ as described, and an extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$. Since E is the smallest set containing $\iota(T)$ closed under $\bar{\oplus}, \bar{\odot}$ and $\bar{\otimes}$, it can be shown, that $E = \iota(D) \cup \{F_{\overline{\mathsf{kill}}}\}$, with $F_{\overline{\mathsf{kill}}}(x) = x \odot \mathsf{kill}$, and $\eta(F_f) = f$ and $\eta(F_{\overline{\mathsf{kill}}}) = \mathsf{kill}$.

This is can be explained by the fact, that a kill occurring in a parallel has the most impact on the result if it is executed as last transition, where it can not be overwritten. Thus we need an additional weight, that describes exactly the effect, that once a kill has occurred as the result of a parallel process, it has to always be the last weight considered. The function $F_{\overline{\mathsf{kill}}}$, describing a partial tree containing a kill as the result of a parallel branch, does exactly that.

In contrast to kill, id and gen influence an interleaving the most if they are considered as early as possible. In this case the weight of the parallel branch needs to be considered right after it was created. Thus parallel composition degenerates to sequential composition and no additional information needs to be stored.

Example 7. To determine, whether the value of a written in line 2 is always used in the calculation in line 3, we can use a forward must bitvector analysis. We set the weights for the transitions of the DPN to be $f(r_4) = \mathsf{gen}$, i.e. if we encounter the write at line 2, we set the bit to 1, $f(r_6) = \mathsf{kill}$, i.e. we set the bit to 0 if line 7 writes, and $f(r) = \mathsf{id}$ for all other rules. If the resulting function sets the bit, we know, that the write in line 2 is always the last write to a before the use in line 3.

KILL/GEN. KILL/GEN analyses are a special kind of dataflow analysis, where dataflow facts are elements of a complete distributive lattice $(D, \sqcup)$, with least and greatest elements $\bot, \top$, and the set of transformers is restricted to $T = \{f : D \to D \mid \exists k, g \in D \text{ with } f(x) = (x \sqcap k_f) \sqcup g_f\}$. They can be used for bitvector analyses, but also encompass other analyses, like strong copy constant propagation [6].

We only consider forward KILL/GEN analyses, but backward analyses can be handled similarly. The semiring is $\mathcal{S} = (T \cup \{\mathsf{zero}\}, \oplus, \odot, \mathsf{zero}, \mathsf{id})$, where zero is an artificial element representing the zero element of the ring and $\mathsf{id}(x) = (x \sqcap \top) \sqcup \bot$. For elements $f, g \neq \mathsf{zero}$ we then have $(f \oplus g)(x) = f(x) \sqcup g(x)$ and $(f \odot g)(x) = g(f(x))$. In [6] it was shown, that $f \otimes g = f \odot g \oplus g \odot f$ is an abstract interleaving operator.

Applying the construction described above, we arrive at an extended semiring $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, F_{\mathsf{zero}}, F_{\mathsf{id}})$ and extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$. With $\iota(g)(f)(x) = f(g(x))$ it can be shown, that every element $F \in E \setminus \{F_{\mathsf{zero}}\}$ can be written as $F(f)(x) = f(f_F(x)) \sqcup i_F$ with $f_F \in T, i_F \in D$. The operations on the extended semiring can then be reduced to operations of the semiring and underlying lattice as follows:

$$\begin{aligned}
(F \mathbin{\bar{\odot}} G)(f)(x) &= f((f_F \odot f_G)(x)) \sqcup (i_F \sqcup i_G) \\
(F \mathbin{\bar{\oplus}} G)(f)(x) &= f((f_F \oplus f_G)(x)) \sqcup (i_F \sqcup i_G) \\
(F \mathbin{\bar{\otimes}} G)(f)(x) &= f(f_G(x \sqcap k_{f_F})) \sqcup (i_F \sqcup i_G \sqcup g_{f_F}) \\
\eta(F)(x) &= f_F(x) \sqcup i_F
\end{aligned}$$

On the one hand the result for the interleaving operator can be seen as generalisation of the bitvector result. Parallel effects that improve the result are applied as early as possible, directly on the initial information and effects that worsen the result are applied as late as possible, after all information has been

computed. On the other hand we arrive at a result similar to [8]. Here it was observed, that KILL/GEN analyses can be solved by separating paths directly reaching a program point from the possible interference of the environment. The same structure can be found in the extended weight domain, where a weight is described by a standard transformer, representing the reaching path, that is applied to the initial data and a lattice element, representing the possible interference, that is added at the end.

Example 8. To determine, which writes of a can be used in the calculation in line 3, we can use a forward KILL/GEN analysis over the lattice $(2^{\{2,7\}}, \subseteq)$. We set the weights for the transitions of the DPN to be $f(r_4) = \lambda x.\{2\}$, $f(r_6) = \lambda x.\{7\}$ and $f(r) = \lambda x.x$ for all other rules. If we apply the resulting function to the empty set, we get the set of writing locations whose value can be used in line 3.

5 Solving the BGPP Problem for BWDPN

Now consider an execution hedge in $\mathsf{Hedges}(C_1, C_2)$. Each tree of the hedge transforms a stack in a starting configuration $c_1 \in C_1$ into a configuration containing the transformed original stack and stacks of spawned processes, that is part of a target configuration $c_2 \in C_2$. Analogous to the approach in [2], we can split each tree into several parts along the rightmost branch. We differentiate between two main types of partial trees. The first type transforms an initial stacktop of the form $p\gamma$ into cp', meaning that the topmost stacksymbol is popped off the stack. The second type transforms $p\gamma$ into $cp'w$, with $w \in \Gamma^+$, pushing additional symbols on the stack. In both cases new process may be spawned and transformed, forming the configuration c to the left of the initial process. If we take an execution tree τ, we can observe, that the execution of the initial process can always be split into a sequence of pop transformation and a final push transformation.

If we now classify the partial trees by their initial stack $p\gamma$ and their result cp' or $cp'w$, we can assemble each execution tree out of these classes. Taking for example τ_1, τ_2 with $p_i\gamma_i \overset{\tau_i}{\Longrightarrow} c_i p_{i+1}$ and τ_3 with $p_3\gamma_3 \overset{\tau_3}{\Longrightarrow} p'w'$, we get an execution tree $\tau = \tau_1 ; \tau_2 ; \tau_3$ with $p_1\gamma_1\gamma_2\gamma_3 w \overset{\tau}{\Longrightarrow} c_1c_2c_3p'w'w$.

Since the spawned processes and pushed stacksymbols of a partial tree are unbounded this is still an infinite number of classes. We exploit the fact, that we are only interested in the trees that reach a given regular set of configurations and assume the set is described by an automaton. The spawned processes and pushed stacksymbols of a partial tree will not be altered by a concatenated tree, it will only spawn and transform its own new processes and a push is the final phase of an execution. Consequently the spawned processes of a partial tree and the symbols pushed onto the stack have to be part of the final configuration. Since the configuration is part of a regular set we can describe these parts by two states of the automaton between which a part is accepted. Grouping the classes where the spawned processes and pushed stacksymbols are accepted by the same states together, we arrive at a finite number of classes.

To characterise these classes, we take a closer look at the saturation procedure introduced in [1] to compute the set of predecessor configurations of a given target set.

Regular Sets of Configurations. The saturation procedure requires special kinds of automata for representation of the target set. We use $\mathcal{M}$- and $\mathcal{M}^*$-automata, adapted from [1], as a compact representation for the target set. A $\mathcal{M}^*$-automaton is a finite automaton $\mathcal{A}^* = (S, P \cup \Gamma, \delta, \dot{s}, F)$ that satisfies the following additional conditions:

- $S^C, S^P \subseteq S$, where for all $s \in S^C, p \in P$ exists a unique and distinguished state $s_p \in S^P$
- $\delta = \delta^P \cup \delta^\Gamma$ where $\delta^P = \{(s, p, s_p) \mid s \in S^C, p \in P\}$ and $\delta^\Gamma \subseteq S \times (\Gamma \cup \{\varepsilon\}) \times S$
- $\mathcal{L}(\mathcal{A}) \subseteq \mathsf{Conf}$

A $\mathcal{M}$-automaton $\mathcal{A}$ is a $\mathcal{M}^*$-automaton, where the transition relation δ satisfies the stronger condition $\delta^\Gamma \subseteq S \times (\Gamma \cup \{\varepsilon\}) \times (S \setminus S^P)$ and $\dot{s} \in S \setminus S^P$. We write $s \xrightarrow{\lambda}_\delta s'$ for $(s, \lambda, s') \in \delta$ and $s \xrightarrow{c}_\delta^* s'$ for the reflexive transitive closure. $\mathcal{L}(\mathcal{A})$ is the language of the automaton. Each regular set of configurations can be described by an $\mathcal{M}$-automaton. For the rest of the paper we fix two $\mathcal{M}$-automata $\mathcal{A}_1 = (S_1, P \cup \Gamma, \delta_1, \dot{s}_1, F_1)$ and $\mathcal{A}_2 = (S_2, P \cup \Gamma, \delta_2, \dot{s}_2, F_2)$ with $\mathcal{L}(\mathcal{A}_1) = C_1$ and $\mathcal{L}(\mathcal{A}_2) = C_2$.

Characterising Trees and Hedges. The following saturation procedure, taken from [1], works by adding new transitions to the automaton $\mathcal{A}_2$, thus allowing more configurations to be accepted. The result is a $\mathcal{M}^*$-automaton $\mathcal{A}_2^* = (S_2, P \cup \Gamma, \bar{\delta}_2, \dot{s}_2, F_2)$, with $\bar{\delta}_2 = \delta_2^P \cup \bar{\delta}_2^\Gamma$, where $\bar{\delta}_2^\Gamma$ is the smallest set fulfilling the conditions:

$$
\begin{array}{ll}
[\mathsf{init}] & t \in \bar{\delta}_2^\Gamma \text{ if } t \in \delta_2^\Gamma \\
[\mathsf{step}] & (s_p, \gamma, s') \in \bar{\delta}_2^\Gamma \text{ if } r = p\gamma \hookrightarrow c \in \Delta, s \in S^C, s \xrightarrow{c}_{\bar{\delta}_2}^* s'
\end{array}
$$

A transition is added, if there is a rule transforming the symbol into a configuration which is accepted by previously existing transitions. If these transitions were also added by the saturation, they themselves have a rules, which transform their symbols. If we follow this recursion and assemble the rules into a tree, we have a tree that transform the symbol of the newly added transition into a configuration that can be read using only transition of $\mathcal{A}_2$. Consequently all new configurations $\mathcal{L}(\mathcal{A}_2^*)$ which are accepted because of this transition, are predecessors of configurations in the original automaton. Additionally a new transition (s_p, γ, s') is a witness for the existence of a tree, that transforms $p\gamma$ into a configuration c which is accepted between the states s and s'. If $s' \in S^P$ then $c = c'p'$, since only P transitions reach states in S^P and the tree is a pop transformation as described above. If $s' \notin S^P$, we have $c = c'p'w'$ and the tree is a push transformation.

We later extend the saturation procedure to collect all of these trees for a transition by constructing a constraint system L over $(2^{\mathsf{Trees}}, \cup)$, similar to the

grammar used to describe executions in [2]. The variables of the constraint system $\mathsf{L}[t]$ with $t \in \bar{\delta}_2^\Gamma$ can be seen as annotations to the transitions of the saturated automaton. The least solution of the constraint system then corresponds exactly to the classes of trees described above. We define a function $\pi_\mathsf{L} : S_2 \times \mathsf{Conf} \times S_2 \to 2^{\mathsf{Hedges}}$ that constructs a set of hedges for a configuration by reading the annotations from the automaton $\mathcal{A}_2^*$:

$$
\begin{aligned}
[\mathsf{empty}]\quad & \pi_\mathsf{L}(s,\varepsilon,s) && = \{\varepsilon_\sigma\} \\
[\mathsf{epsilon}]\quad & \pi_\mathsf{L}(s,\varepsilon,s') && = \bigcup\{\pi_\mathsf{L}(s,\varepsilon,s'') \mid s'' \xrightarrow{\varepsilon}_{\bar{\delta}_2^\Gamma} s'\} \\
[\mathsf{control}]\quad & \pi_\mathsf{L}(s,cp,s') && = \bigcup\{\pi_\mathsf{L}(s,c,s'')\varepsilon_\tau \mid s'' \xrightarrow{p}_{\bar{\delta}_2^P} s'\} \\
& && \quad \cup \bigcup\{\pi_\mathsf{L}(s,cp,s'') \mid s'' \xrightarrow{\bar{\varepsilon}}_{\bar{\delta}_2^\Gamma} s'\} \\
[\mathsf{stack}]\quad & \pi_\mathsf{L}(s,c\gamma,s') && = \bigcup\{\pi_\mathsf{L}(s,c,s'')\,;\mathsf{L}[(s'',\gamma,s')] \mid s'' \xrightarrow{\gamma}_{\bar{\delta}_2^\Gamma} s'\} \\
& && \quad \cup \bigcup\{\pi_\mathsf{L}(s,c\gamma,s'') \mid s'' \xrightarrow{\varepsilon}_{\bar{\delta}_2^\Gamma} s'\}
\end{aligned}
$$

If we read a partial configuration c between two states of the saturated automaton, we can construct the set of hedges transforming the configuration into a partial configuration of the target set accepted between the same two states, using π_L.

We start with the set containing only the empty hedge and add a new empty tree, whenever we read a control state of the DPN. Consider for example now a configuration $p\gamma_1\gamma_2$. After reading p we are in a state s_p and the current set of hedges is $\{\varepsilon_\tau\}$. ε transitions do not contain any additional information and information is simply propagated trough. If we now read the next symbol in the saturated automaton, we can distinguish two cases for the next transition:

Either the transition is $(s_p, \gamma_1, s'_{p'})$. As observed above all trees annotated to this transition are pop transformations, applying them to $p\gamma_1\gamma_2$ ends in a configuration $cp'\gamma_2$, where $s \xrightarrow{c}{}^*_{\delta_2} s'$. The next transition for γ_2, is then $(s'_{p'}, \gamma_2, s'')$, which is annotated with trees transforming $p'\gamma_2$ into configurations c' with $s' \xrightarrow{c'}{}^*_{\delta_2} s''$. If we concatenate the trees, we get transformations of $p\gamma_1\gamma_2$ to cc', with $s \xrightarrow{cc'}{}^*_{\delta_2} s''$.

Or the transition is (s_p, γ_1, s'), with $s' \notin S^P$. As observed the trees annotated to this transition are push transformations. Starting from $p\gamma_1\gamma_2$, they lead to $cp'w'\gamma_2$, with $s \xrightarrow{cp'w'}{}^*_{\delta_2} s'$. Since we can accept the configuration as part of the target set, we have another transition (s', γ_2, s'') in the original automaton. The set of trees transforming the configuration $p\gamma_1\gamma_2$ is then simply the set of the first transition. To simplify the construction, we annotate transitions not starting in states in S^P, with the set only containing the empty tree, thus we can simply concatenate the annotated sets of all transitions in both cases.

This can be extended to a configuration with an arbitrary number of stack symbols. If we take a configuration $p_1w_1p_2w_2$, we want to construct the set of hedges transforming the configuration. If we read the configuration left to right we construct the set of trees for the first stack a described above. If we now encounter a transition for a control state, a new initial empty tree is added to the end of the hedges. We can then construct the set of trees for second stack

again as described above, since all concatenation operations now concern this new last tree of the hedges. This can be extended to a configuration with an arbitrary number of stacks.

Using these observations, we construct a set of constraints in a similar way the saturation procedure adds transitions to the automaton:

$$[\mathsf{init}] \qquad \mathsf{L}[t] \supseteq \{\varepsilon_\tau\} \text{ if } t \in \delta_2^\Gamma$$
$$[\mathsf{step}]\ \mathsf{L}[(s_p, \gamma, s')] \supseteq \{r(\varepsilon_\tau)\} ; \pi_{\mathsf{L}}(s, c, s') \text{ if } r = p\gamma \hookrightarrow c \in \Delta, s \in S^C, s \xrightarrow{c}{}^*_{\bar{\delta}_2} s'$$

The trees are essentially constructed bottom up. Each transition starts with the set containing only the empty tree. If we add a new transition, we add all trees which can be constructed by the rule which lead to the addition, and all hedges, which are already known to transform the configuration reached by that rule.

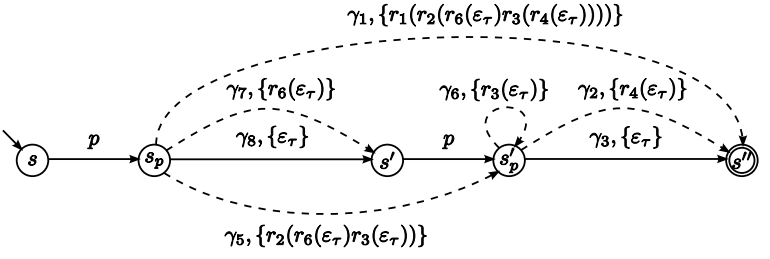

Fig. 5. Annotated automaton after saturation

Figure 5 shows part of the resulting automaton of the saturation procedure applied to the target C_2 set of our example and the least solution of the constraint system annotated to the transitions of the automaton. The initial automaton is displayed with solid arrows and transitions added by the saturation are dashed.

For the least solution $\mathsf{lfp}(\mathsf{L})$ of L we can prove, by induction on the structure of the trees τ:

Lemma 9. *For $s \in S_2^C, s' \in S_2, p \in P, \gamma \in \Gamma, (s_p, \gamma, s') \in \bar{\delta}_2^\Gamma$, we have:*

$$\mathsf{lfp}(\mathsf{L})[(s_p, \gamma, s')] = \{\tau \mid \exists c \in \mathsf{Conf} \text{ with } p\gamma \overset{\tau}{\Longrightarrow} c, s \xrightarrow{c}{}^*_{\delta_2} s'\},$$

and for $s \notin S_2^P, s' \in S_2, \gamma \in \Gamma, (s, \gamma, s') \in \bar{\delta}_2^\Gamma$, we get:

$$\mathsf{lfp}(\mathsf{L})[(s, \gamma, s')] = \{\varepsilon_\tau\}.$$

Thus the solution of the constraint system contains exactly the classes of trees we wanted to characterise. If we annotate the transitions of $\mathcal{A}_2^*$ with $\mathsf{lfp}(\mathsf{L})$, we can prove by induction on the length of the configurations c:

Lemma 10. *For $s, s' \in S_2, c \in \mathsf{Conf}$, we have:*

$$\pi_{\mathsf{lfp}(\mathsf{L})}(s, c, s') = \{\sigma \mid \exists c' \in \mathsf{Conf} \text{ with } c \overset{\sigma}{\Longrightarrow} c', s \xrightarrow{c'}{}^*_{\delta_2} s'\},$$

Hence we have $\mathsf{Hedges}(\{c\}, C_2) = \bigcup\{\pi_{\mathsf{lfp}(\mathsf{L})}(\dot{s}_2, c, s) \mid s \in F_2\}$ for a configuration $c \in \mathsf{Conf}$. We can describe the set of all reaching hedges from the single configuration c into the set C_2. We are now interested in the union of all these sets for configurations in C_1. It suffices to consider configurations in $C_1 \cap \mathcal{L}(\mathcal{A}_2^*)$, since all other configurations have an empty set of reaching hedges. Since the number of configurations can still be infinite and thus we can not evaluate $\pi_{\mathsf{lfp}(\mathsf{L})}$ for all these configurations, we construct a second constraint system O over $(\mathsf{Hedges}, \cup)$, that imitates the computation of $\pi_{\mathsf{lfp}(\mathsf{L})}$ by propagating sets of hedges along the transitions of an automaton and joining preliminary results at each state of the automaton.

Since we only want the result for configurations in $C_1 \cap \mathcal{L}(\mathcal{A}_2^*)$, we construct the constraint system for the product automaton $\mathcal{A}_3 = (S_3, P \cup \Gamma, \delta_3, \dot{s}_3, F_3)$ of $\mathcal{A}_1$ and $\mathcal{A}_2^*$, describing the intersection. For $s \in S_3$ we write $s|_i$, with $i \in \{1, 2\}$, to refer to the original state of automaton $\mathcal{A}_i$ that was used to form s in the product automaton.

$$
\begin{array}{lll}
[\mathsf{empty}] & \mathsf{O}[\dot{s}] \supseteq \{\varepsilon_\sigma\} & \\
[\mathsf{epsilon}] & \mathsf{O}[s'] \supseteq \mathsf{O}[s] & \text{if } (s, \varepsilon, s') \in \delta_3^\Gamma \\
[\mathsf{control}] & \mathsf{O}[s'] \supseteq \mathsf{O}[s]\varepsilon_\tau & \text{if } (s, p, s') \in \delta_3^P \\
[\mathsf{stack}] & \mathsf{O}[s'] \supseteq \mathsf{O}[s]\,;\mathsf{lfp}(\mathsf{L})[(s|_2, \gamma, s'|_2)] & \text{if } (s, \gamma, s') \in \delta_3^\Gamma
\end{array}
$$

Since it works on the transitions of the product automaton, the constraint system emulates the steps of $\pi_{\mathsf{lfp}(\mathsf{L})}$ on $\mathcal{A}_2^*$ and simultaneously ensures, that each transition followed in $\mathcal{A}_2^*$ has a corresponding transition in $\mathcal{A}_1$. Thus it only works on configurations which are also in C_1.

It can then be shown, by induction on the length of the configuration c, that:

Lemma 11. *For $s \in S_3$, we get:*

$$\mathsf{lfp}(\mathsf{O})[s] = \bigcup\{\pi_{\mathsf{lfp}(\mathsf{L})}(\dot{s}_3|_2, c, s|_2) \mid \dot{s}_3|_1 \xrightarrow{c}{}^*_{\delta_1} s|_1\},$$

Consequently, the solution of the constraint system at the accepting states of the product automaton can be used to describe the set of all connecting hedges:

Theorem 12. *We get:*

$$\mathsf{Hedges}(C_1, C_2) = \bigcup\{\mathsf{lfp}(\mathsf{O})[s] \mid s \in F_3\}.$$

Abstraction. To compute the weight of the hedges, we construct a constraint system $\mathsf{L}^\#$, a function $\pi^\#_{\mathsf{L}^\#}$ and constraint system $\mathsf{O}^\#$ over the weight domain by replacing the operators and constants in the constraint system L, the function π_L and constraint system O, with the corresponding operators and constants according to the abstraction function β:

$$
\begin{array}{cc}
(2^{\mathsf{Hedges}}, \cup) \rightsquigarrow (E, \bar{\oplus}) & M \rightsquigarrow \beta(M) \\
M\,;M' \rightsquigarrow \beta(M) \mathbin{\bar{\odot}} \beta(M') & MM' \rightsquigarrow \beta(M) \mathbin{\bar{\otimes}} \beta(M')
\end{array}
$$

Since the order in the abstract domain is dual to the ordering on sets of hedges, we compute the greatest fixpoint in the abstract domain. Using standard results from abstract interpretation [9], we get $\mathsf{gfp}(\mathsf{L}^\#) = \beta(\mathsf{lfp}(\mathsf{L}))$, $\pi^\#_{\mathsf{gfp}(\mathsf{L}^\#)} = \beta \circ \pi_{\mathsf{lfp}(\mathsf{L})}$ and $\mathsf{gfp}(\mathsf{O}^\#) = \beta(\mathsf{lfp}(\mathsf{O}))$ for the solutions of the constraint systems. With Theorem 12:

Theorem 13. *It follows, that:*

$$\theta(C_1, C_2) = \bar{\bigoplus}\{\mathsf{gfp}(\mathsf{O}^\#)[s] \mid s \in F_3\}.$$

Thus we can solve the BGPP problem by computing $\mathsf{gfp}(\mathsf{L}^\#)$ and $\mathsf{gfp}(\mathsf{O}^\#)$. Theorem 5 states, that we get the solution to the GPP problem by applying η.

Algorithm. Given a WDPN $(\mathcal{M}, \mathcal{S}, f)$ and two sets of configurations C_1, C_2, represented by $\mathcal{M}$-automata $\mathcal{A}_1, \mathcal{A}_2$, the complete algorithm to compute the solution of the GPP problem $\delta(C_1, C_2)$ consists of the following steps:

1. Find a suitable extended semiring $\mathcal{E}$ and extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$ and consider the BWDPN $(\mathcal{M}, \mathcal{E}, \bar{f})$, with $\bar{f}(r) = \iota(f(r))$.
2. Construct the automaton $\mathcal{A}_2^*$ using the saturation procedure. The saturation can be done in $O(|S_2|^3|\Delta|\|\Delta\|)$ time, where $\|\Delta\|$ is the length of the longest right hand side of a rule in Δ. The size of the transition relation of the saturated automaton is in $O(|\delta_2| + |S_2|^2|\Delta|)$.
3. Construct the abstract constraint system $\mathsf{L}^\#$ for $\mathcal{A}_2^*$ and solve it. The construction can be done during the saturation of the automaton. The size of the constraint system is in $O(|\delta_2| + |S_2|^{2\|\Delta\|}|\Delta|)$. The time needed to solve the constraint system depends on the solver and the height and complexity of the weight domain.
4. Compute the product automaton $\mathcal{A}_3$ from $\mathcal{A}_1$ and $\mathcal{A}_2^*$.
5. Construct the abstract constraint system $\mathsf{O}^\#$ for $\mathcal{A}_3$ and solve it. The construction can be done during the computation of the product automation. The size of the constraint system is equal to the size of the transition relation of the product automaton and thus in $O(|\delta_1|(|\delta_2| + |S_2|^2|\Delta|))$.
6. Compute $\bar{\bigoplus}\{\mathsf{gfp}(\mathsf{O}^\#)[s] \mid s \in F_3\}$.
7. Apply η to get $\delta(C_1, C_2)$.

In total the algorithm is linear in the size of the program $|\Delta|$, exponential in the size of the rules $\|\Delta\|$ and polynomial in the number of states and transitions of the automata describing the starting and target sets of the query. Since all DPN can be transformed into DPN with only rules of the type $p\gamma \hookrightarrow p'$, $p\gamma \hookrightarrow p'\gamma'$, $p\gamma \hookrightarrow p'\gamma'\gamma''$ and $p\gamma \hookrightarrow p'\gamma'p''\gamma''$, where the number of rules increases by a constant factor, the size of the rules $\|\Delta\|$ can be considered fixed and small. Similarly the starting and target sets of a query are usually representable by small automata and thus we have an efficient algorithm. For the first two applications described in Section 4 the solution of the constraint system can be computed using standard fixpoint algorithms. Termination of the computation is guaranteed, since the domains do not contain infinite descending chains. For the KILL/GEN analyses we additionally require, that the underlying lattice has no infinite ascending chains, to ensure termination of the fixpoint iteration.

6 Conclusion

We presented the GPP problem for a WDPN, which is a model for parallel programs with dynamic process creation and recursive procedures. The GPP problem is a general problem formulation, which can, for example, be used to capture basic dataflow analysis problems. Since the GPP problem can not be solved directly, our approach is based on an alternative branching semantics for DPN. The resulting tree shaped executions can be characterised using a constraint system, which can then be solved over an abstract domain to get a solution for the BGPP problem for BWDPN. If the weight domains for the BWDPN and WDPN are connected through an extension, the solution for the GPP problem can be derived from the corresponding BGPP problem. We have shown how the results can be used to solve basic dataflow analysis problems like bitvector analyses or shortest path problems.

Future Work. Firstly, we are currently working on an implementation of the algorithm and different weight domains.

Another direction of research is the iterated application of our algorithm. One can observe, that the product automaton computed in Section 5 is again an $\mathcal{M}$-automaton, whose transitions are indirectly annotated by the solution of the constraint system. The idea is to take this automaton as the target set for a second computation, which is initialized with the annotation of the automaton. Similar techniques have been studied for DPN without weights [11] and WPDS [4] to realize context-bounded analyses.

To compute the BGPP solution we need to solve a constraint system over the extended semiring. In practice this requires the extended semiring to fulfill additional criteria for the computation to terminate, like finiteness or the descending chain condition. To deal with unbounded domains, widening [9] could be introduced. Additionally in recent work [12,13], new techniques have been presented to solve equations for more general types of semirings. We plan on examining, whether these can be applied to our extended semirings.

In addition the relation between a semiring and a corresponding extended semiring and extension needs to be studied further. Here especially conditions which guarantee the existence of an efficient construction are of interest. Or alternatively, whether there are ways to construct at least an efficient approximation.

Our main application of BWDPN up to now is solving the GPP problem for WDPN. BWDPN themselves can be interesting. One example are weight domains which rely on thread identity. The thread executing a specific transition can not be determined from an interleaved path, but is visible in an execution hedge. The acquisition structures studied in [10], to compute whether there exists a path connecting two regular sets of configurations w.r.t. a lock sensitive semantics, can for example be adapted into a weight domain for BWDPN. Furthermore we plan to investigate whether the approach to the analysis of synchronisation taken in [14] can be adapted to our framework and thus extended to dynamic process creation.

Acknowledgement. We thank Markus Müller-Olm and Peter Lammich for helpful discussions on the topic of dynamic pushdown networks.

References

1. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)
2. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2) (2005)
3. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2) (2000)
4. 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 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
5. Esparza, J., Podelski, A.: Efficient algorithms for pre^* and $post^*$ on interprocedural parallel flow graphs. In: POPL. ACM, New York (2000)
6. Seidl, H., Steffen, B.: Constraint-based inter-procedural analysis of parallel programs. Nordic J. of Computing 7(4) (2000)
7. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: efficient and optimal bitvector analyses for parallel programs. ACM Trans. Program. Lang. Syst. 18(3) (1996)
8. Lammich, P., Müller-Olm, M.: Precise fixpoint-based analysis of programs with thread-creation and procedures. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 287–302. Springer, Heidelberg (2007)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. ACM, New York (1977)
10. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 525–539. Springer, Heidelberg (2009)
11. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
12. Esparza, J., Kiefer, S., Luttenberger, M.: Newton's method for ω-continuous semirings. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 14–26. Springer, Heidelberg (2008)
13. Kühnrich, M., Schwoon, S., Srba, J., Kiefer, S.: Interprocedural dataflow analysis over weight domains with infinite descending chains. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 440–455. Springer, Heidelberg (2009)
14. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. ACM, New York (2003)