A note on implementing the admissible semantics

`DungAF`'s implementation of the admissible semantics essentially comprises three stages —

1. The first stage finds the *defence-sets* around each argument. An argument's defence-sets are those sets of arguments which (i) include the argument; and (ii) are admissible; and (iii) subsume no other set satisfying (i) and (ii) [1].

2. The second stage uses the set of all defence-sets to find the preferred extensions.

3. The third stage uses the preferred extensions to find the remaining (non-minimal, non-maximal) non-empty admissible sets, if any exist.

While the second and third stages are straightforward, the first stage is less so. `DungAF` uses a simplified and slightly modified version of Vreeswijk's algorithm to determine the defence-sets around an argument [1]. Vreeswijk's algorithm was designed to generate *labelled* defence-sets[1], and includes a slight error; `DungAF`'s version is simplified in that it generates merely defence-sets, and is modified to address the error.

`DungAF`'s algorithm is given below. Vreeswijk provides the outline of a proof for his algorithm [1], and this proof is substantially adequate for the modified version of the algorithm. Furthermore, the test classes `TestWithAspartix` and `TestDefenceSets` suggest that the modified version (as implemented by the method `getDefenceSetsAround(String)`) is correct. However, a separate sketched proof is also given below.

Algorithm 1: ***modifiedVreeswijk06(...)***

Let *target-arg* be the argument whose defence-sets are sought. *modifiedVreeswijk06* takes the following parameters:

1. *arg-fram* — an argumentation framework;

2. *path* — a list of arguments;

3. *can-sols* — a set of candidate-solutions. Loosely speaking, a candidate-solution *cs* is such that the algorithm has not yet established that *cs* is neither (a) a defence-set around *target-arg*; nor (b) a strict subset of such a defence-set.

4. *current-arg* — either *target-arg*; or an argument *arg* which attacks some argument in every $cs \in$ *can-sols*, and is not attacked by any $cs \in$ *can-sols*; or, given such an argument *arg*, an argument *arg*$'$ which attacks *arg*, and is such that for every $cs \in$ *can-sols*, $(cs \cup \{arg'\})$ is conflict-free.

[1]The label denotes, in each case, whether the defence-set is merely an admissible set, or an admissible set which is also a subset of the grounded extension.

Algorithm 1: modifiedVreeswijk06(arg-fram, current-arg, path, can-sols)

```
input  : The argumentation framework arg-fram; the argument current-arg; the list of arguments path; the set
         of argument-sets can-sols.
output: The defence-sets around current-arg in arg-fram.
1  on-pro-arg = path is even-length;            // find whether current-arg is a pro-arg or opp-arg.
   // If can-sols is empty, there are no candidate-solutions to consider, so return $\emptyset$. Otherwise
      if current-arg attacks itself, it cannot be used to expand any $S \in$ can-sols to form an
      admissible set, so return $\emptyset$, if on-pro-arg.
2  if either can-sols = $\emptyset$ or (on-pro-arg & current-arg attacks itself in arg-fram) then
3      return $\emptyset$;
4  end
5  add current-arg to path;
6  if on-pro-arg then add current-arg to each $S \in$ can-sols;
7  else accumulated-can-sols = $\emptyset$;
   // Recurse for arguments attacking current-arg, as appropriate.
8  foreach argument attacker attacking current-arg in arg-fram do
9      relevant-can-sols = $\emptyset$;
10     if on-pro-arg then self-defensive-can-sols = $\emptyset$;
11     foreach $S \in$ can-sols do
12         if on-pro-arg then
13             if no arg $\in S$ attacks attacker in arg-fram then add $S$ to relevant-can-sols;
14             else add $S$ to self-defensive-can-sols;
15         end
16         else
17             if ($S \cup \{$attacker$\}$) is conflict-free in arg-fram then add $S$ to relevant-can-sols;
18         end
19     end
       // If on-pro-arg, redefine can-sols...
20     if on-pro-arg then
21         can-sols = modifiedVreeswijk06(arg-fram, attacker, path, relevant-can-sols);
22         add all of self-defensive-can-sols to can-sols;
23         remove all non-minimal members of can-sols;
24         if can-sols is empty then break;                 // not necessary, but sensible.
25     end
       // ...otherwise augment accumulated-can-sols.
26     else
27         add all of modifiedVreeswijk06(arg-fram, attacker, path, relevant-can-sols) to accumulated-can-sols;
28         remove all non-minimal members of accumulated-can-sols;
29     end
30 end
31 if not on-pro-arg then can-sols = accumulated-can-sols;
32 remove current-arg from path;
33 return can-sols;
```

An external call to the algorithm supplies as parameters (i) the relevant argumentation framework, (ii) the empty list, (iii) $\{\emptyset\}$ and (iv) *target-arg*. In the interval between the calling and return of the externally-called instance, a complete or partial depth-first exploration of the tree rooted in *target-arg* in *arg-fram* is performed, and the sets returned by the externally-called instance are derived from that exploration. The algorithm recurses with (some or all) steps taken downwards through the tree. In each recursively-called instance, *arg-fram* is the argumentation framework which was passed to the calling instance; *current-arg* is the argument down to which the search has stepped; *path* is the list of arguments which had been passed to the calling instance, extended by the argument which had been passed to the calling instance; and *can-sols* is an appropriately updated version of the set of candidate-solutions which had been passed to the calling instance.

Each instance of the algorithm proceeds as follows. First it finds whether *current-arg* is a pro-arg or an opp-arg — i.e. whether, if added to *path*, it would have the same parity as the first argument, which must be *target-arg*, and hence would intuitively be pro-*target-arg*, given that every argument in *path* attacks its predecessor. If *current-arg* is an opp-arg, the instance is *on-opp-arg*; otherwise it is *on-pro-arg* (line 1). Except in trivial scenarios (lines 2–4), the algorithm then proceeds as follows. It first adds *current-arg* to *path*, in order to ensure that recursively-called instances of the algorithm are correctly classified as *on-pro-arg* or *on-opp-arg* (line 5). If this instance is *on-pro-arg*, *current-arg* is added to every member of *can-sols*; otherwise an empty set is prepared to store new candidate-solutions (lines 6–7). The algorithm then recurses for some or all of the arguments attacking *current-arg* (lines 8–30). The arrangement of this recursion and the treatment of the output of each called instance of the algorithm depends on whether the calling instance is *on-pro-arg* or *on-opp-arg*.

Suppose first that the calling instance is *on-pro-arg*. If *current-arg* = *target-arg*, *can-sols* contains just {*target-arg*}, and this recursive stage just continues the process of setting in motion the series of recursions which will eventually produce *current-arg*'s defence-sets. Otherwise *current-arg* is an argument which defends each member of *can-sols* against some argument *arg′* which attacks an argument common to all of them; *current-arg* is not itself in conflict with with any of those candidate-solutions, and hence each of the latter remains conflict-free upon the addition of *current-arg* (line 6). The recursive stage therefore addresses a particular sub-task — to set in motion the series of recursions which will eventually determine which (if any) of the members of *can-sols* are suitably defended by *current-arg* against *arg′*. Since *current-arg* is not in conflict with any $\textit{can-sol}_\text{i} \in \textit{can-sols}$, its suitability as a defender of each $\textit{can-sol}_\text{i} \in \textit{can-sols}$ against *arg′* is in question in only one respect — whether there is a defence-set around *current-arg*, such that its union with $\textit{can-sol}_\text{i}$ is conflict-free[2].

Let *attacker* be an argument attacking *current-arg* (line 8); let *self-defensive-can-sols* be those members of *can-sols* which attack *attacker*; and let *relevant-can-sols* =

[2]Of course, even if $\textit{can-sol}_\text{i}$ can be expanded to form a conflict-free set which subsumes a defence-set around *current-arg*, it need not also be a subset of any of the sought defence-sets. With respect to the latter issue, the suitability of *current-arg* as a defender of each $\textit{can-sol}_\text{i} \in \textit{can-sols}$ against *arg′* is in question in a broader respect — whether there is a defence-set around *current-arg*, such that its union with $\textit{can-sol}_\text{i}$ is (strictly or non-strictly) subsumed by any of the sought defence-sets.

(*can-sols*$\setminus$ *self-defensive-can-sols*) (lines 9–15). *relevant-can-sols* therefore comprises those members of *can-sols* rendered inadmissible by *attacker*; it is not yet clear that those members cannot be expanded to form defence-sets, but if they are to be so expanded, we must look more closely at the place of *attacker* in *arg-fram*. Hence the algorithm recurses for *attacker*, supplying *arg-fram*, the (now extended) *path*, *relevant-can-sols* and *attacker* to the called instance.

The calling instance then redefines *can-sols* as the output of the called instance. It then adds all of the *self-defensive-can-sols* to *can-sols*, and finally removes the non-minimal members of *can-sols* (lines 21–23). It might be that these latter are subsumed by defence-sets around *target-arg*, but they can be safely discarded, for the retention of their minimal counterparts ensures that any such defence-sets will be found anyway.

If *can-sols* is now empty, the recursive stage of the algorithm ceases, even if there remain arguments attacking *current-arg* for which the algorithm has not yet recursed (line 24). For the emptiness of *can-sols* implies that there is no defence against *attacker* which suits any of the candidate-solutions which were passed to the calling instance — i.e. that none of those candidate-solutions can be expanded to form any admissible set which includes *current-arg*. There is therefore no need to consider any further arguments attacking *current-arg*.

If, on the other hand, *can-sols* is not empty, the algorithm recurses for some other attacker of *current-arg* for which the algorithm has not yet recursed. Therefore in the course of the recursive stage, *can-sols* might fluctuate considerably, now increasing, now decreasing in cardinality. In the end, *can-sols* will not include any of the candidate-solutions which were passed to the calling instance, but each of its members will subsume at least one of those candidate-solutions. Each might substantially subsume its counterpart S, but might not do so — if (and only if) S was such that $(S \cup \{\textit{current-arg}\})$ defended *current-arg* against all attacks, would $(S \cup \{\textit{current-arg}\}) \in \textit{can-sols}$ at line 33.

Suppose now that the calling instance is *on-opp-arg*. *current-arg* not merely attacks each member of *can-sols* (by virtue of attacking an argument common to all of them), but is not itself attacked by any of them, and hence renders them inadmissible. This recursive stage therefore sets in motion the series of recursions which will eventually determine, for each $\textit{can-sol}_i \in \textit{can-sols}$, which of the arguments attacking *current-arg* are suitable defenders of $\textit{can-sol}_i$ against *current-arg* — that is, which of them have at least one defence-set such that its union with $\textit{can-sol}_i$ is conflict-free.

Again let *attacker* be an argument attacking *current-arg* (line 8). This time let *relevant-can-sols* be those members of *can-sols* which are not in conflict with *attacker* — i.e. those members for which *attacker* is not immediately obviously unsuitable as a defender against *current-arg* (line 11; lines 16–18). The calling instance recurses, supplying *arg-fram*, the (now extended) *path*, *relevant-can-sols* and *attacker* to the called instance, and adds the output to *accumulated-can-sols* (line 27).

As would not necessarily be the case were the calling instance *on-pro-arg*, the calling instance recurses for all arguments attacking *current-arg* — i.e. regardless of whether any called instance returns the empty set. For the correctness of the algorithm demands that *all* possible defenders of the members of *can-sols* against *current-arg* be considered. The output of the called instances — *accumulated-can-sols* — can, however, be safely filtered to remove the non-minimal members (line 28), as the retention of their minimal counterparts guarantees that correctness is not thereby endangered.

After its recursive stage, then if the algorithm is *on-opp-arg*, it redefines *can-sols* as *accumulated-can-sols* (line 31). Regardless of whether the instance is *on-opp-arg*, the algorithm must remove *current-arg* from *path*, to ensure that subsequent instances are correctly classified as *on-pro-arg* or not (line 32). It then returns *can-sols* (line 33).

Correctness of *modifiedVreeswijk06(. . .)* — a sketch

Now let us informally show that the algorithm is correct. We first show that an external call to the algorithm returns only defence-sets around the specified argument, and then that it returns all such sets.

modifiedVreeswijk06(. . .) returns only defence-sets

Admissibility requires of a set of arguments (i) that it is conflict-free and (ii) that it is acceptable with respect to itself — i.e. that it attacks every argument which attacks any of its members. Therefore to show that an external call to the algorithm returns only defence-sets of the specified argument, we must show that every returned set (i) contains the specified argument; (ii) is conflict-free; (iii) is acceptable with respect to itself; and (iv) is minimal among the sets satisfying (i)–(iii). For the sake of clarity and brevity, we henceforth refer to instances of the algorithm which are *on-opp-arg* and instances which are *on-pro-arg* as *opp-instances* and *pro-instances* respectively.

It is straightforward to show that every set returned by the algorithm contains the specified argument. Every instance of the algorithm returns *can-sols*; every pro-instance adds *current-arg* to every member of *can-sols*; and nowhere does the algorithm reduce any set of arguments. In an externally-called instance of the algorithm, *current-arg* is the specified argument, and externally-called instances are pro-instances. Therefore every set of arguments returned by an externally-called instance contains the specified argument.

It is also straightforward to show that every set returned by the algorithm is conflict-free. Three points to do with the expansion of sets of arguments are important — (i) that such expansion happens only in pro-instances; (ii) that the sets expanded are invariably members of the set of candidate-solutions passed to the instance; and (iii) that the manner of expansion is invariably merely the addition of the argument that has been passed to the instance. If a pro-instance is the externally-called instance, the set of candidate-solutions passed to it contains only $\emptyset$, which of course remains conflict-free after the addition of any argument. If a pro-instance is not the externally-called instance, it must have been called by an opp-instance; and no opp-instance passes to a pro-instance a set of arguments and an argument, such that adding the latter to the former produces a non-conflict-free set. Therefore in no instance of the algorithm is a set of arguments expanded to form a non-conflict-free set. Furthermore, no instance of the algorithm creates a set of arguments *anew* (i.e. not by expanding an existing set). Therefore sets of arguments which are not conflict-free nowhere feature, and hence cannot be returned.

Now let us consider acceptability. We show by contradiction that every set of arguments returned by an externally-called instance of the algorithm is acceptable with respect to itself. Let S be a set of arguments which is not acceptable with respect to itself, on account of (i) including an argument *undefended-arg*, and (ii) not attacking an argument *attacker* which attacks *undefended-arg.* Suppose now that S is among the argument-sets returned by an externally-called instance of the algorithm.

As we have seen, the externally-called instance of the algorithm begins with no sets of arguments except the empty set, and sets of arguments are expanded only in pro-instances, and only by the addition of the argument which was passed to the pro-instance. Therefore were the algorithm to return S, there must have been some pro-instance $inst_{\text{pro}}$, which had among its parameters (i) *undefended-arg* and (ii) a set of candidate-solutions which included a candidate-solution S' such that $S' \subset S$. Since $S' \subset S$, $(S' \cup \{undefended\text{-}arg\})$ does not defend *undefended-arg* against *attacker.*

Now, there might be multiple arguments attacking *undefended-arg*, and $inst_{\text{pro}}$ recurses for each, each opp-instance being passed those members of *can-sols* which do not defend themselves against the attacker, and *can-sols* being redefined after each opp-instance has returned as the minimal members of the union of (i) that instance's output and (ii) the candidate-solutions which were found to defend themselves against the attacker. However, whichever order in which the attackers of *undefended-arg* are thereby dealt with, when the algorithm recurses for *attacker* the opp-instance $inst_{\text{attacker}}$ will be passed those members of *can-sols* which do not defend themselves against *attacker.* $inst_{\text{attacker}}$ is an opp-instance, so for any candidate-solution returned by $inst_{\text{attacker}}$, that candidate-solution must have been returned by a pro-instance, such that the argument passed to it attacks *attacker.* Now, in the course of that pro-instance every candidate-solution which was passed to it must have been expanded by the addition of the argument that was passed to it — i.e. the argument attacking *attacker.* Nowhere does the algorithm reduce any candidate-solution; therefore every candidate-solution returned by $inst_{\text{attacker}}$ contains an argument attacking *attacker.* Therefore when $inst_{\text{attacker}}$ returns and $inst_{\text{pro}}$ then redefines *can-sols* as the minimal members of the union of (i) the output of $inst_{\text{attacker}}$ and (ii) those candidate-solutions which were found to defend themselves against *attacker*, *can-sols* contains no candidate-solution which does not defend itself against *attacker.* Therefore, again since the algorithm nowhere reduces any candidate-solution, $inst_{\text{pro}}$ cannot return any candidate-solution which does not defend itself against *attacker.* Therefore, in sum, if *undefended-arg* is added to any candidate solution, some attacker of *attacker* must also be added to that same candidate solution, if the externally-called instance is to return the candidate-solution. Therefore S cannot be among the argument-sets returned by the externally-called instance of the algorithm.

Let us finally consider minimality. Every set returned by the algorithm is minimal, because no instance returns any set of candidate-solutions which has not first been filtered to remove the non-minimal members — such filtering is inevitably the last thing to happen in the recursive stage (lines 23 and 28), and no sets of arguments are expanded or created thereafter.

modifiedVreeswijk06(...) returns all defence-sets

Having shown that an external call to the algorithm generates only defence-sets for the specified argument, let us now show that it generates all of those defence-sets. To do so, we must show that every defence-set around *target-arg* is both found and retained — i.e. that in some instance the set is constructed, and that the externally-called instance returns the set. We consider these matters in turn for an arbitrary such set $S^{\text{defence}}_{\text{target-arg}}$.

That $S^{\text{defence}}_{\text{target-arg}}$ will be found follows from the manner in which the algorithm recurses and the manner in which it expands candidate-solutions. Let us show this by tracking its construction, referring to its partial, in-progress versions as cs' throughout. In the externally-called instance, the set of candidate-solutions is initially $\{\emptyset\}$, and will become $\{\{\textit{target-arg}\}\}$ before any recursion. So $cs' = \{\textit{target-arg}\}$. If and only if *target-arg* is attacked does the algorithm recurse. Suppose that b is among *target-arg*'s attackers, and that the algorithm recurses first of all for b; and suppose also that c attacks b and $c \in S^{\text{defence}}_{\text{target-arg}}$. The admissibility of $S^{\text{defence}}_{\text{target-arg}}$ implies that $\{\textit{target-arg}, c\}$ is conflict-free, so when the instance recurses for b, the called-instance will recurse for c — i.e. passing c and $\{\textit{target-arg}\}$ to the called instance, which we term $inst_{\text{c}}$. In $inst_{\text{c}}$ cs' is expanded to form $\{\textit{target-arg}, c\}$. If cs' defends c against all attackers, $inst_{\text{c}}$ returns $\{cs'\}$. If it does not, let d be the first uncountered attacker for which recursion occurs. The process is then repeated, except that this time d takes the place of b, and an argument e in $S^{\text{defence}}_{\text{target-arg}}$ which attacks d takes the place of c. And so on, until cs' is augmented by an argument k which is adequately defended by $(cs' \cup \{k\})$. This augmentation occurs, of course, in a pro-instance; that instance returns to its calling instance a set which includes cs'. Now, that calling-instance might then recurse many more times, for alternative defenders against the attack which motivated the inclusion of k; however, cs' is in any case preserved unchanged in *accumulated-can-sols*, to be returned to the calling instance, unless a proper subset of it is added to *accumulated-can-sols*. Assume for the moment that no such subset is added, and that cs' is returned to the calling instance. The calling instance is of course a pro-instance, the *pro-arg* in question being in cs'. That instance might then discard cs' as non-minimal; again, assume for the moment that it does not. If there remain attackers of *pro-arg* which have not been dealt with, cs' will be expanded by another series of instances, as described above. Otherwise cs' is returned to the calling instance, which returns it to the previous pro-instance, again assuming that it is not discarded as non-minimal. And so on, until $S^{\text{defence}}_{\text{target-arg}}$ has been found.

We have so far assumed that the construction of $S^{\text{defence}}_{\text{target-arg}}$ proceeds straightforwardly — i.e. that none of the in-progress versions denoted by cs' is discarded as non-minimal. Let us now show that even if cs' — that is, an in-progress version of $S^{\text{defence}}_{\text{target-arg}}$ — is ever discarded, $S^{\text{defence}}_{\text{target-arg}}$ must be discovered nonetheless, and returned by the externally-called instance. This fact follows from (i) the manner in which the algorithm *discards* candidate-solutions; and (ii) the fact that it returns only defence-sets of *target-arg*; and (iii) the fact that the algorithm invariably terminates.

In pro-instances, candidate-solutions are discarded only if they are non-minimal. In opp-instances, candidate-solutions might be discarded in one of two ways. Either (a) the candidate-solution is one of those which was passed to the instance, and there is no attacker of the argument which was passed to the instance which is not in conflict with the candidate-solution; or (b) the candidate-solution is one which was returned by one of the called pro-instances, and it turns out to be non-minimal in the union of the outputs of the called instances. Therefore whenever a candidate-solution is discarded, a candidate-solution subsumed by it is necessarily retained — that is, returned to the calling instance or passed to a called instance — unless it is discarded in manner (a) in an opp-instance. Now, (a) cannot be true of cs', because cs' is a subset of an admissible set.

Therefore if cs' is ever discarded, the instance in which it is discarded returns or passes some subset cs'' to another instance; and if cs'' was then discarded in that other instance, then some subset of it would be returned or passed... and so on, until we reach a subset which is not discarded. What becomes of it? Because the algorithm returns only defence-sets of *target-arg*, it cannot be returned, since it is a proper subset of $S^{\text{defence}}_{\text{target-arg}}$, and hence inadmissible. Therefore either it is expanded to form $S^{\text{defence}}_{\text{target-arg}}$, or the algorithm does not terminate.

Let us therefore show that the algorithm invariably terminates. The algorithm might fail to terminate only if the recursion might be either infinitely broad or infinitely deep — i.e. if it might be that any instance either recursed infinitely or called one instance, which called another, which called another... *ad infinitum.* Recursion is finite in terms of breadth, for each instance may recurse n times at most, where n is the number of arguments attacking the argument which was passed to it; and no argument has infinitely many attackers. Recursion is finite also in terms of depth. Note first that such a series of instances as sketched must be alternately *on-opp-arg* and *on-pro-arg.* In every pro-instance, the candidate-solutions which are passed to that instance are augmented with the argument which was passed to the instance; it passes a subset of those candidate-solutions to the next instance in the series. That opp-instance does not expand the candidate-solutions passed to it, but nor does it reduce them — it simply passes a subset of them to the next (pro-) instance in the series. Therefore in such a series of instances as was sketched, increasingly large candidate-solutions are passed to the instances as the series progresses. Therefore such a series of instances cannot continue *ad infinitum*, as there are only finitely many arguments. Therefore recursion is finite in terms of depth as well as breadth, and the algorithm invariably terminates.

References

[1] G. A. W. Vreeswijk. An algorithm to compute minimally grounded and admissible defence sets in argument systems. In P. E. Dunne and T. J. M. Bench-Capon, editors, *Proceedings of the 1st International Conference on Computational Models of Argument (COMMA'06)*, pages 109–120, 2006.