2014 IEEE 26th International Conference on Tools with Artificial Intelligence

ARGTEACH – A learning tool for Argumentation Theory

Jeremie Dauphin, Claudia Schulz
Department of Computing
Imperial College London
London, UK
jeremie.dauphin09,claudia.schulz@imperial.ac.uk

***Abstract*—Following the increasing influence of the formal study of argumentation in AI research, Abstract Argumentation (AA) is now taught as part of Computer Science degrees at various universities. To support the teaching of AA we present ARGTEACH, an interactive intelligent tutor that facilitates the learning of different labelling semantics in AA. The user assigns the labels `in`, `out`, and `undec` to arguments in an AA framework displayed as a graph, with the aim to find all complete labellings. The user then determines which of the complete labellings are grounded, preferred, semi-stable, or stable. During the labelling process, ARGTEACH supports the user by providing hints about possible next labelling steps, using a novel method for computing complete labellings, and by checking whether the labelling done so far is correct.**

***Keywords*-Computer Science Education, Electronic Learning, Logic**

I. INTRODUCTION

During the past 20 years, the formal study of argumentation has received increasing attention in the field of Artificial Intelligence (AI). Intuitively, an *argument* is a statement, a reason for or against a decision, or even a belief. For example, "Let's go for a walk" is an argument and so is "It's raining outside, we'll get wet", where the latter is a counter-argument of the first, i.e. the second argument *attacks* the first. In this scenario it is easy to determine that the second argument is a "winning argument" as it is not attacked, and that therefore the first argument is a "losing argument". However, in more complex scenarios involving more arguments and attacks, deciding on the winning and losing arguments is often less straight forward.

Thus, in AI the idea of arguments and attacks has been characterized in a formalism called *Abstract Argumentation* (AA) [1], whose various *semantics* provide different ways of determining winning and losing arguments. Some semantics adopt a sceptical view whereas others follow a credulous approach, so that depending on the chosen semantics different sets of arguments may be deemed to be winning and losing. The relationship between the different semantics as well as their properties have been extensively studied in recent years [2], [3], [4], and the different semantics have been found suitable for different application purposes [5], [6], [7], demonstrating the importance of studying and being able to distinguish all the different semantics.

Due to its increasing popularity as a reasoning method in AI, AA is no longer merely a subject of research, but is now taught at various universities as part of the Computer Science curriculum: The TU Dresden offers a Seminar on AA[1], the University of Potsdam a course on Formal Models of Argumentation[2], and Imperial College London a lecture on Argumentation and Multi-Agent Systems[3]. The University of Dundee is even planning to create three MSc degrees dealing with argumentation[4]. So far, the only tool designed to support teaching of AA is Araucaria [8], where the user has to identify arguments as well as attacks between them from text. However, Araucaria does not support determining which arguments are winning and losing according to the different semantics of AA. On the other hand, ASPARTIX [9] and similar implementations (see Section VII) compute winning and losing arguments for a given AA framework according to the different semantics, but do not provide any explanation on how they were determined.

To support the teaching of AA at universities as well as to assist novices in familiarising with the semantics of AA we present ARGTEACH[5], an interactive software which facilitates the user's learning in how to determine sets of winning and losing arguments according to the different AA semantics by providing the user with hints, explanations, and error-checking. For this purpose, ARGTEACH makes use of two commonly used methods in AA: Firstly, arguments are displayed as nodes in a graph and attacks as edges between these nodes. Secondly, sets of winning and losing arguments are determined by assigning one of the labels `in` (for winning), `out` (for losing), and `undec` (for neither winning nor losing) to each argument according to labelling rules [10]. There are often various ways to assign these labels according to the rules, resulting in different possible sets of winning and losing arguments. ARGTEACH combines these two methods in an easy-to-use graphical interface, where the user assigns labels to arguments step-by-step and receives feedback on the current progress if required:

[1] http://www.inf.tu-dresden.de/?node_id=3457&ln=en
[2] http://www.cs.uni-potsdam.de/wv/lehre/05WS/05-FormModArg/
[3] http://www3.imperial.ac.uk/computing/teaching/courses/474
[4] http://www.arg-tech.org/index.php/teaching/postgraduate-study-in-persuasion-negotiation-and-critical-thinking/
[5] http://www.doc.ic.ac.uk/~cis11/ArgTeach/argTeach.html

1082-3409/14 $31.00 © 2014 IEEE
DOI 10.1109/ICTAI.2014.120

If at any point the user is unsure how to continue the labelling, ARGTEACH will provide hints about possible next labelling steps. Furthermore, if the user is not sure whether the assignment of labels done so far is correct, ARGTEACH can check for mistakes. Being an interactive (intelligent) tutor [11], ARGTEACH can facilitate the user's learning as well as considerably improve his/her understanding about the different semantics of AA.

II. BACKGROUND

An *Abstract Argumentation Framework* (AAF) [1], [10] is a tuple $\langle Ar, Att \rangle$, where Ar is a set of *arguments* and $Att \subseteq Ar \times Ar$ is a set of *attacks* between them. $(A, B) \in Att$ means that argument A attacks argument B. We equivalently say that B is attacked by A, and that A is an attacker of B. Given sets of arguments $Args_1, Args_2 \subseteq Ar$:

- $Args_1$ *attacks* B iff some argument $A \in Args_1$ attacks B. $Args_1$ attacks $Args_2$ iff $Args_1$ attacks some argument $B \in Args_2$.
- $Args_1^+ = \{A \in Ar \mid Args_1 \text{ attacks } A\}$.
- $Args_1$ is *conflict-free* iff it does not attack itself.
- $Args_1$ *defends* an argument $A \in Ar$ iff $Args_1$ attacks all arguments $B \in Ar$ attacking A. $Args_1$ defends $Args_2$ iff $Args_1$ defends every argument in $Args_2$.

From now on, we will assume as given some AAF $\langle Ar, Att \rangle$. The semantics of an AAF are determined by assigning one of the labels `in`, `out`, and `undec` to each argument, where `in` denotes winning arguments, `out` losing ones, and `undec` neutral arguments which are neither winning nor losing.

A *total labelling* [10] is a total function $Lab : Ar \to \{\texttt{in}, \texttt{out}, \texttt{undec}\}$.

The set of arguments labelled `in` under Lab is denoted $\texttt{in}(Lab)$; the sets of arguments labelled `out` and `undec` are denoted $\texttt{out}(Lab)$ and $\texttt{undec}(Lab)$, respectively. The different semantics of an AAF impose different restrictions on total labellings [10]. The most important total labelling is the complete labelling, as all other labelling semantics considered here can be determined based on the complete labelling.

A total labelling Lab is a *complete labelling* iff for every argument $A \in Ar$:

- If A is labelled `in` then all B attacking A are labelled `out`;
- If A is labelled `out` then some B attacking A is labelled `in`;
- If A is labelled `undec` then neither of the above applies, i.e. then A is attacked by some B labelled `undec` and all C attacking A are labelled `undec` or `out`.

Note that a total labelling Lab can also be shown to be complete by proving that the three conditions are satisfied the other way around, i.e. when substituting the "if" with "only if" in all three conditions.

A complete labelling Lab is:

- the *grounded labelling* iff $\texttt{in}(Lab)$ is minimal (w.r.t. set inclusion) among all complete labellings;
- a *preferred labelling* iff $\texttt{in}(Lab)$ is maximal (w.r.t. set inclusion) among all complete labellings;
- a *semi-stable labelling* iff $\texttt{undec}(Lab)$ is minimal (w.r.t. set inclusion) among all complete labellings;
- a *stable labelling* iff $\texttt{undec}(Lab) = \emptyset$.

Originally, the semantics of an AAF were defined as *extensions* [1], i.e. sets of "winning arguments" satisfying certain conditions. We will here only mention the complete extension: A conflict-free set of arguments $Args$ is a *complete extension* iff it consists of all arguments it defends. Note that there is a direct correspondence between labellings and extensions [2]: $Args$ is a complete extension iff Lab with $\texttt{in}(Lab) = Args$, $\texttt{out}(Lab) = Args^+$, and $\texttt{undec}(Lab) = Ar \backslash (Args \cup Args^+)$ is a complete labelling.

III. PRELIMINARY DEFINITIONS

Since ARGTEACH provides a platform for the step-by-step labelling of initially unlabelled arguments, most of the computation involves dealing with situations where some arguments already have a label, but others are still unlabelled. We call such labellings "partial labellings".

Definition 1 (Partial Labelling)**.** A *partial labelling* is a partial function $Lab_{part} : Ar \to \{\texttt{in}, \texttt{out}, \texttt{undec}\}$.

Analogously to total labellings, $\texttt{in}(Lab_{part})$, $\texttt{out}(Lab_{part})$, and $\texttt{undec}(Lab_{part})$ denote the sets of arguments labelled `in`, `out`, and `undec`, respectively. In addition, $unlab(Lab_{part})$ is the set of arguments left unlabelled by Lab_{part}. Note that total labellings are a special kind of partial labellings, where $unlab(Lab_{part}) = \emptyset$. We sometimes represent a partial labelling as a set of argument-label pairs of the form $Lab_{part} = \{(A, \texttt{in}), \ldots, (E, \texttt{out}), \ldots, (H, \texttt{undec})\}$, which does not comprise pairs for unlabelled arguments.

Since the aim of the ARGTEACH user it to find complete labellings, we are particularly interested in partial labellings which so far satisfy the conditions of a complete labelling.

Definition 2 (Correct Partial Labelling)**.** A partial labelling Lab_{part} is a *correct partial labelling* iff for every $A \in Ar$:

- If A is labelled `in` then all B attacking A are labelled `out`;
- If A is labelled `out` then some B attacking A is labelled `in`;
- If A is labelled `undec` then A is attacked by some B labelled `undec` and all C attacking A are labelled `undec` or `out`.

In order to provide hints in ARGTEACH, we need to check whether or not the current partial labelling can be

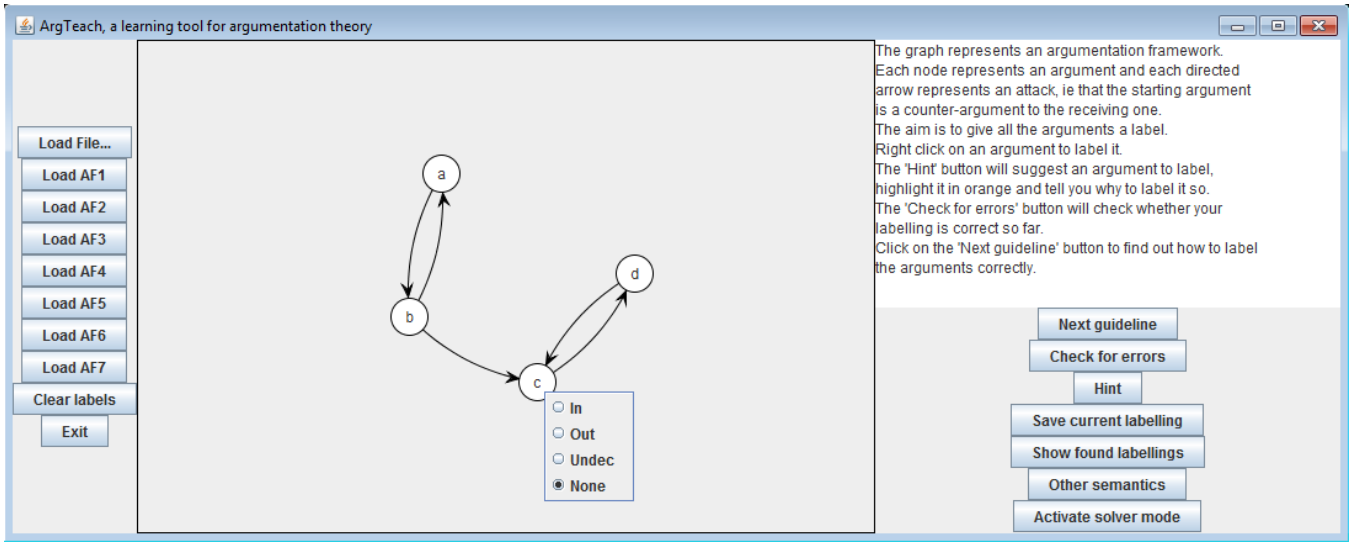

Figure 1. The main user interface of ARGTEACH displaying the predefined $AF5$ with all arguments unlabelled.

extended to another partial (or even a complete) labelling by leaving all labels assigned so far as they are and assigning appropriate labels to all unlabelled arguments.

Definition 3 (Extending a partial labelling)**.** Let Lab_{part_1} and Lab_{part_2} be partial labellings. *Lab_{part_1} can be extended to Lab_{part_2}*, denoted $Lab_{part_1} \subseteq Lab_{part_2}$, iff there exist disjoint sets of arguments $X_{\texttt{in}}, X_{\texttt{out}}, X_{\texttt{undec}} \subseteq unlab(Lab_{part_1})$ such that:

- $\texttt{in}(Lab_{part_2}) = \texttt{in}(Lab_{part_1}) \cup X_{\texttt{in}}$;
- $\texttt{out}(Lab_{part_2}) = \texttt{out}(Lab_{part_1}) \cup X_{\texttt{out}}$;
- $\texttt{undec}(Lab_{part_2}) = \texttt{undec}(Lab_{part_1}) \cup X_{\texttt{undec}}$.

Equivalently, we say that *Lab_{part_2} is reachable from Lab_{part_1}*. In Section V we show that if a partial labelling is correct, it can be extended to a complete labelling using the algorithm for providing hints in ARGTEACH.

IV. ARGTEACH

ARGTEACH is compatible with both Windows and Linux; in both cases SWI-Prolog and Java RE have to be installed. For a given AAF the user has to find all complete labellings by assigning one of the labels `in`, `out`, or `undec` to every argument. Subsequently, the user has to identify which of these complete labellings are grounded, preferred, semi-stable, or stable. The main user interface of ARGTEACH is split into three panels as displayed in Fig.1.

A. The left and central panel

In the left panel, the user can choose from one of the seven example AAFs or load his/her own AAF into ARGTEACH. Note that ARGTEACH abbreviates an Abstract Argumentation Framework as "AF" instead of "AAF". An AAF can be easily defined in a text file by characterizing every argument $A \in Ar$ as $arg(a)$, and every attack $(A, B) \in Att$ as $att(a, b)$, where argument names have to start with a lower case letter.

The central panel displays the chosen AAF as a graph, where arguments are nodes and attacks are directed edges between these nodes. Initially, all arguments are white, indicating that they are unlabelled. The user can change the label of an argument infinitely many times, including unlabelling the argument, by selecting the desired label from the selection box appearing upon right-clicking on the argument (see Fig.1). When labelling an argument as `in`, `out`, or `undec` its colour changes to green, red, or grey, respectively.

B. The right panel

The top part of the right panel serves as a user-guide, explaining how to use ARGTEACH on the first "page" and outlining the rules of a complete labelling on the following "pages". By clicking the 'Next guideline' button, the user can scroll through the different "pages" of the user-guide.

The lower part of the right panel comprises the two main teaching features of ARGTEACH: If the user is unsure which argument to label next, ARGTEACH can provide hints about possible next steps along with a reason for the suggestion. Furthermore, the user can check if the labels assigned so far are correct using the 'Check for errors' button.

To help the user find all complete labellings the user can save the current partial labelling at any point, given that it is correct. If the current labelling is in fact a complete labelling, it is saved as a found complete labelling. At any point, the user can see the complete labellings found so far as well as the last partial labelling saved by clicking the 'Show found labellings' button. Any saved labelling can then be loaded into the central panel, overriding the currently displayed labelling.

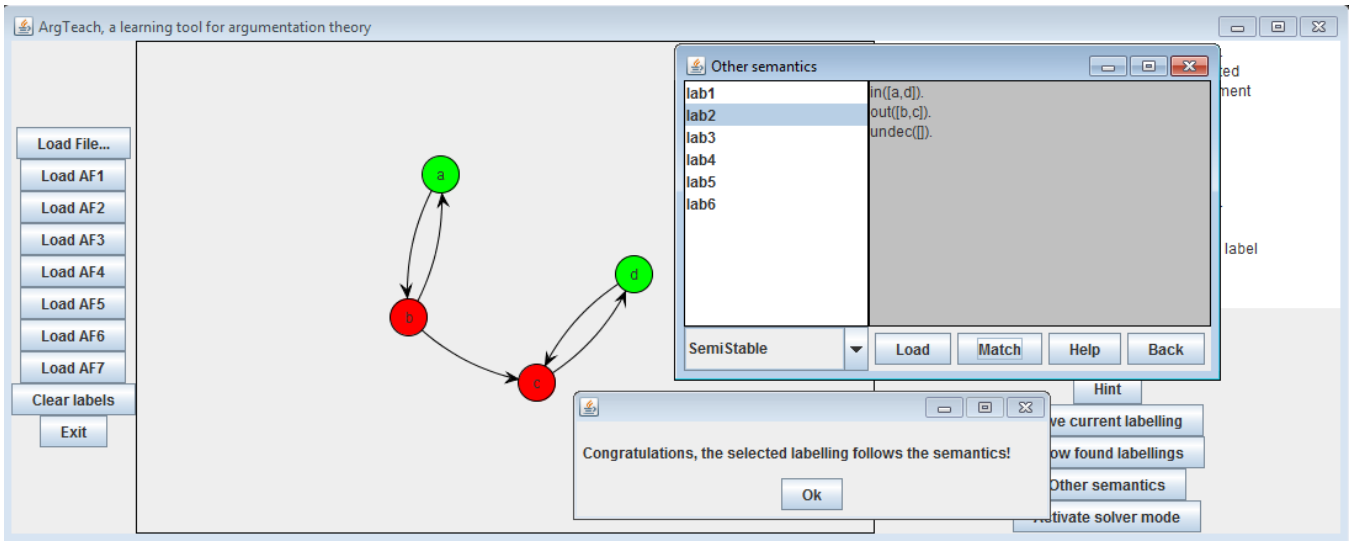

Figure 2. The 'Other semantics' interface of ARGTEACH. The user correctly matched $lab2$, which was previously loaded into the central panel, as a semi-stable labelling.

Once the user has found all complete labellings, the 'Other semantics' button can be used to identify which of the complete labellings are grounded, preferred, semi-stable, or stable. As shown in Fig.2, the complete labellings can be loaded into the central panel if the user requires a graphical representation, and can be matched with a semantics selected from the drop-down menu. ARGTEACH will then give feedback on whether or not the match is correct.

As an additional feature, ARGTEACH can compute all complete labellings for the user. However, since ARGTEACH is intended for learning purposes, this feature should not usually be used by learners. Clicking the 'Activate solver mode' button will therefore prompt a warning asking if the user really wants to activate the solver mode. After agreeing and restarting ARGTEACH, clicking the 'Solve' button now substituting the 'Activate solver mode' button will cycle through all complete labellings displaying them in the central panel.

V. PROVIDING HINTS

One of the main teaching features of ARGTEACH is its ability to provide *hints* about possible next labelling steps. Based on the current partial labelling, ARGTEACH suggests a label for an unlabelled argument along with an explanation for the suggestion. In many situations, there are different possible next labelling steps, which are all pointed out by ARGTEACH. Note that upon pressing the 'hint' button, ARGTEACH does not generally check whether the current partial labelling is correct, meaning that ARGTEACH provides hints as if the current partial labelling could be extended to a complete labelling. This design was chosen to focus the user's attention on the steps involved in the labelling process and to support learning from mistakes [12].

After introducing the hint algorithm in more detail in the next section, we will prove that it exhibits two important properties: Firstly, if the user follows the hints in every step right from the start, the resulting total labelling will be a complete labelling no matter which of the hints the user chooses in each step (Theorem 3). Secondly, every complete labelling of an AAF can be found by following a certain sequence of hints (Theorem 4).

A. The nextHint Algorithm

Hints are computed in Prolog and then handed over to a Java wrapper. Alg.1 gives an outline of the nextHint predicate, which provides different hints for a given partial labelling on backtracking.

Alg.1 distinguishes five hint-scenarios: It first checks whether the given partial labelling Lab_{part} is a complete labelling (line 1) using the complete(Lab_{part}) predicate. This predicate is implemented according to the definition of complete extension, i.e. by creating conflict-free sets of arguments consisting of all arguments the set defends. If the current partial labelling is indeed a complete labelling, no backtracking for further hints will occur since the Java wrapper will not ask for any further solutions. Otherwise, Alg.1 looks for unlabelled arguments which should be labelled `out` because they are attacked by some `in` argument (line 2), and for arguments which should be labelled `in` because they are only attacked by arguments labelled `out` (line 3). Iteratively applying this method of assigning `in` and `out` straight from the beginning and then assigning the label `undec` to still unlabelled arguments yields the grounded labelling [2], [13].

However, since the user should be able to find all complete labellings using the hints, we use an additional novel

Algorithm 1 hints for a partial labelling Lab_{part}

1: **nextHint**(Lab_{part}, 'Done', 'Done', 'Done') :-
complete(Lab_{part}).
2: **nextHint**(Lab_{part}, A, 'Out', 'attacked by some In') :-
$A \in unlab(Lab_{part})$,
$\exists B : (B, A) \in Att \wedge B \in \texttt{in}(Lab_{part})$.
3: **nextHint**(Lab_{part}, A, 'In', 'only attacked by Out') :-
$A \in unlab(Lab_{part})$,
$\forall B : (B, A) \in Att \rightarrow B \in \texttt{out}(Lab_{part})$.
4: **nextHint**(Lab_{part}, A, 'In', 'defends itself') :-
$Args \subseteq unlab(Lab_{part})$, $A \in Args$,
$newIn = Args \cup \texttt{in}(Lab_{part})$,
$\nexists B \in \texttt{undec}(Lab_{part})$:
$(B, C) \in Att \wedge C \in Args$,
conflictfree($newIn$), defends($newIn$, $Args$),
not defends($\texttt{in}(Lab_{part})$, $Args$),
$\nexists Args' \subset Args : Args' \neq \emptyset \wedge$
defends($\texttt{in}(Lab_{part}) \cup Args'$, $Args'$).
5: **nextHint**(Lab_{part}, A, 'Undec', 'neither In nor Out') :-
$A \in unlab(Lab_{part})$, complete(Lab),
$Lab_{part} \subset Lab$, $A \in \texttt{undec}(Lab)$, !.

method for assigning the label `in` to unlabelled arguments: Alg.1 finds minimal (non-empty) sets of unlabelled arguments which are not defended by $\texttt{in}(Lab_{part})$, but can defend themselves against all attackers when combined with $\texttt{in}(Lab_{part})$ (line 4). If such a set is not attacked by any argument already labelled `undec` (all attackers of `in` arguments have to be `out`), ARGTEACH suggests to label one argument in this *in-self-defending set* as `in`. Due to this novel labelling strategy of in-self-defending sets, all complete labellings of an AAF can be found using the hints provided by ARGTEACH (see Theorem 4).

Example 1. Consider the AAF in Fig.3 with the partial labelling $Lab_{part} = \{(d, \texttt{in})\}$. The set $\{f, h\}$ is not defended by $\texttt{in}(Lab_{part})$, but by $\texttt{in}(Lab_{part}) \cup \{f, h\}$ since $\{f, h\}$ defends h from its only attacker g and f from the attacker e, and since $\texttt{in}(Lab_{part})$ defends f from its second attacker a. $\{f, h\}$ is minimal since neither f nor h can defend itself, and thus $\{f, h\}$ is a n in-self-defending set.

The last hint-scenario (line 5) is concerned with labelling an unlabelled argument A as `undec` if the current partial labelling Lab_{part} can be extended to a complete labelling Lab where $A \in \texttt{undec}(Lab)$. Note that for all but the first and last hint-scenario, Alg.1 returns all arguments which can be labelled according to the scenario (on backtracking). In the last hint-scenario only one argument which can be labelled `undec` is returned as based on that further arguments can be labelled `undec` in subsequent steps.

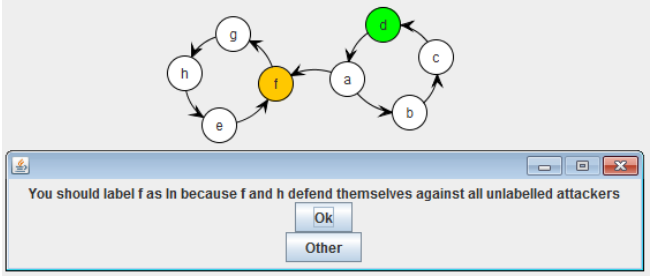

Figure 3. A hint for the predefined $AF7$ involving an in-self-defending set computed by line 4 of Alg.1 (see Example 1).

B. Properties of the nextHint Algorithm

We first prove that if a partial labelling is correct, then Alg.1 will return at least one hint and then this partial labelling can be extended to a complete labelling.

Lemma 1. *Let Lab_{part} be a partial labelling. If Lab_{part} is correct then*

- *nextHint(Lab_{part}, A, Label, Reason) has a solution;*
- *Lab_{part} can be extended to a complete labelling Lab using Alg.1.*

Proof. If Lab_{part} is a correct total labelling, line 1 is applicable, proving both statements. If Lab_{part} is a correct but not complete partial labelling, we show that either lines 2 and 3 or line 5 are applicable. Applying line 2 to all unlabelled arguments and after that applying line 3 to all still unlabelled arguments is equivalent to labelling all arguments attacked by $\texttt{in}(Lab_{part})$ as `out`, yielding ${Lab_{part}}' \supseteq Lab_{part}$, and all arguments defended by $\texttt{in}(Lab_{part})$ as `in`, yielding ${Lab_{part}}'' \supseteq {Lab_{part}}'$. Thus, iteratively applying this procedure yields a labelling $Lab_{part_{fixed}} \supseteq Lab_{part}$ which upon application of lines 2 and 3 does not change anymore. Then $\texttt{in}(Lab_{part_{fixed}})$ consists of all arguments it defends and is therefore a complete extension [1]. Thus, $Lab = Lab_{part_{fixed}} \cup \{(A, \texttt{undec}) \mid A \in unlab(Lab_{part_{fixed}})\}$ is a complete labelling [2]. Consequently, if lines 2 and 3 are not applicable, then $\exists Lab : \forall A \in unlab(Lab_{part}) : A \in \texttt{undec}(Lab)$, and consequently line 5 is applicable.

It follows directly from Lemma 1 that if nextHint fails for a partial labelling, then this labelling is not correct, which ARGTEACH will point out to the user.

Corollary 2. *Let Lab_{part} be a partial labelling. If nextHint(Lab_{part}, A, Label, Reason) fails, then Lab_{part} is not correct.*

We now prove the first important property of Alg.1: If the current partial labelling is correct, then iteratively following a hint computed by Alg.1 will lead to a complete labelling, no matter which of the hints the user follows.

Theorem 3. *Let Lab_{part} be a partial labelling. If Lab_{part} is correct, then following a hint computed by Alg.1 in each*

step will lead to a complete labelling, no matter which hint is chosen in each step.

Proof. We show that after applying a hint computed by lines 2-5 in Alg.1, the new partial labelling can still be extended to a complete labelling by some iterative application of nextHint.

- lines 2 and 3: Satisfy the conditions of a correct partial labelling, so by Lemma 1 the resulting partial labelling can be extended to a complete labelling.
- line 4: The resulting partial labelling is not correct since not all attackers of A are $\texttt{out}$. However, since $newIn$ defends itself, the resulting partial labelling can be extended to a complete labelling. In particular, the iterative application of lines 2 and 3 will lead to a correct partial labelling, which by Lemma 1 can be extended to a complete labelling.
- line 5: Does not always result in a correct partial labelling, but can be extended to a complete labelling as $A \in \texttt{undec}(Lab)$ of some complete labelling $Lab \supset Lab_{part}$.

The next theorem states the second important property of Alg.1: When starting from a partial labelling with all arguments unlabelled, all complete labellings can be found by following a certain sequence of hints.

Theorem 4. *Let Lab_{part} be a partial labelling with all arguments unlabelled. For every complete labelling Lab there exists a sequence of hints computed by Alg.1 leading from Lab_{part} to Lab.*

Proof. Starting from Lab_{part} and iteratively applying lines 2 and 3 until a partial labelling $Lab_{part_{fixed}}$ is reached which does not change anymore, and then iteratively applying line 5 yields the grounded labelling $Lab_{grounded}$ [2], [13]. Let Lab be a complete labelling different from the grounded labelling. Due to the correspondence of labellings and extensions [10], [2] and since the grounded extension is a subset of every complete extension [1], $\texttt{in}(Lab_{grounded}) \subset \texttt{in}(Lab)$. Therefore, some of the arguments labelled $\texttt{undec}$ in $Lab_{grounded}$, i.e. some of the unlabelled arguments in $Lab_{part_{fixed}}$ are labelled $\texttt{in}$ in Lab. Let $Args = \texttt{in}(Lab) \backslash \texttt{in}(Lab_{part_{fixed}})$ be the set of additional $\texttt{in}$ arguments in Lab as compared to $Lab_{part_{fixed}}$. Then $Args \cup \texttt{in}(Lab_{part_{fixed}})$ defends $Args$ and is conflict-free, $Args$ is not attacked by any $\texttt{undec}$ arguments in $Lab_{part_{fixed}}$ (as there are none), and $\texttt{in}(Lab_{part_{fixed}})$ does not defend any $A \in Args$ because otherwise lines 2 and 3 would have labelled A as $\texttt{in}$. Then there is a smallest set $Args' \subseteq Args$ ($Args' \neq \emptyset$) fulfilling these conditions, which will be found by line 4. The other arguments in $Args$ are then found and labelled $\texttt{in}$ either in the same way by line 4 or by the iterative application of lines 2 and 3 until all $A \in Args$ are labelled $\texttt{in}$, and all arguments in $\texttt{out}(Lab)$ are labelled $\texttt{out}$ by line 2. The $\texttt{undec}$ arguments of Lab are then assigned by line 5.

In summary, using the hints provided by ARGTEACH will teach the user how to correctly assign labels to arguments and how to find all complete labellings.

VI. ERROR-CHECKING

The skill of identifying complete labellings does not only involve being able to apply the labelling rules, but also to identify mistakes in a current partial labelling. Therefore, ARGTEACH has an error-checking feature which examines if any of the conditions of a correct partial labelling are violated by the current partial labelling, and if so explains the mistake to the user. In contrast to the hints feature, ARGTEACH only points out one error at a time and does not suggest how to rectify it. As a consequence, correcting an error might not lead to a correct partial labelling since other mistakes may still be present and since the supposed correction undertaken by the user may lead to a new error. This design is intentional to make the user learn from his/her own mistakes [12]. Like the hints feature, the error-checking is implemented in Prolog, examining if the conditions of a correct partial labelling are satisfied by the current partial labelling. ARGTEACH can also identify some further errors, providing the user with more detailed feedback, as outlined in Alg.2 and explained in the following.

To check whether a partial labelling Lab_{part} is correct, Alg.2 examines whether the sets of $\texttt{in}$, $\texttt{out}$, and $\texttt{undec}$ arguments are labelled correctly: Line 3 checks whether the first condition in Def.2 is satisfied, i.e. whether all arguments B attacking an argument $A \in \texttt{in}(Lab_{part})$ are labelled $\texttt{out}$. Lines 5 and 6 check the second condition in Def.2, namely whether every argument $A \in \texttt{out}(Lab_{part})$ is attacked by some argument B labelled $\texttt{in}$. Lines 7 and 8 check the third condition in Def.2, i.e. that an $\texttt{undec}$ argument is not attacked by an $\texttt{in}$ argument and that it is attacked by some $\texttt{undec}$ argument.

In addition to these basic checks, the user is provided with more detailed error-feedback for $\texttt{in}$ and $\texttt{out}$ arguments. The checkOut predicate distinguishes two cases for an $\texttt{out}$ argument A which does not satisfy Def.2: If A is labelled $\texttt{out}$ in some complete labelling reachable from Lab_{part}, line 5 applies and it is pointed out to the user that there is not enough evidence for A to be labelled $\texttt{out}$. If no such complete labelling exists, line 6 applies and ARGTEACH advises that A should not be labelled $\texttt{out}$ as it is not attacked by an $\texttt{in}$ argument. For $\texttt{in}$ arguments, checkIn first verifies that $\texttt{in}(Lab_{part})$ is conflict-free (line 2), as violation of conflict-freeness is the most serious mistake a user could make. If conflict-freeness is satisfied and all attackers of $\texttt{in}$ arguments are labelled $\texttt{out}$, checkIn examines whether $\texttt{in}(Lab_{part})$ is a subset of the $\texttt{in}$ arguments of a complete labelling Lab reachable from Lab_{part}, in particular an *out-maximal* complete labelling, meaning that the overlap of

Algorithm 2 error-checking for a partial labelling Lab_{part}

1: **checkPartialLab**(Lab_{part}, Error, Reason) :-
checkIn(Lab_{part}, Error, Reason);
checkOut(Lab_{part}, Error, Reason);
checkUndec(Lab_{part}, Error, Reason).
2: **checkIn**(Lab_{part}, $Args$, 'can't all be In because not conflict-free') :-
findall(A, $(A \in \mathtt{in}(Lab_{part}), \exists B \in \mathtt{in}(Lab) : (A,B) \in Att \vee (B,A) \in Att)$, $Args$).
3: **checkIn**(Lab_{part}, $Args$, 'should be Out because attacking an In argument') :-
findall(B, $(A \in \mathtt{in}(Lab_{part}), (B,A) \in Att, B \notin \mathtt{out}(Lab_{part}))$, $Args$).
4: **checkIn**(Lab_{part}, $Args$, 'can't be In because attacked by an argument that should be In') :-
complete_maxOutOverlap(Lab, Lab_{part}),
$Args = \mathtt{in}(Lab_{part}) \backslash \mathtt{in}(Lab)$, $Args \neq \emptyset$.
5: **checkOut**(Lab_{part}, A, 'no reason to be Out') :-
$A \in \mathtt{out}(Lab_{part})$,
$\nexists B \in \mathtt{in}(Lab_{part}) : att(B,A)$,
$\exists Lab$: complete(Lab), $A \in \mathtt{out}(Lab)$.
6: **checkOut**(Lab_{part}, A, 'should not be Out because not attacked by an In argument') :-
$A \in \mathtt{out}(Lab_{part})$,
$\nexists B \in \mathtt{in}(Lab_{part}) : att(B,A)$,
$\nexists Lab$: complete(Lab), $A \in \mathtt{out}(Lab)$.
7: **checkUndec**(Lab_{part}, A, 'should not be Undec because attacked by an In argument') :-
$A \in \mathtt{undec}(Lab_{part})$,
$\exists B \in \mathtt{in}(Lab_{part}) : (B,A) \in Att$.
8: **checkUndec**(Lab_{part}, A, 'should not be Undec because not attacked by an Undec argument') :-
$A \in \mathtt{undec}(Lab_{part})$,
$\nexists B \in \mathtt{undec}(Lab_{part}) : (B,A) \in Att$.

$\mathtt{out}(Lab_{part})$ and $\mathtt{out}(Lab)$ is maximal among all reachable complete labellings. If this is the case, the "wrongly $\mathtt{in}$" arguments, which are part of $\mathtt{in}(Lab_{part})$ but not of $\mathtt{in}(Lab)$, are returned as an error (line 4).

Example 2. Consider the AAF and the partial labelling $Lab_{part} = \{(c,\mathtt{in}),(f,\mathtt{in}),(a,\mathtt{out}),(b,\mathtt{out}),(d,\mathtt{out})\}$ shown in Fig. 4. Lines 2 and 3 in Alg.2 both fail, so Prolog tries to prove line 4: complete_maxOutOverlap(Lab, Lab_{part}) gives the out-maximal complete labelling $Lab = \{(a,\mathtt{in}),(c,\mathtt{in}),(e,\mathtt{in}),(g,\mathtt{in}),(b,\mathtt{out}),(d,\mathtt{out}),(f,\mathtt{out}),(h,\mathtt{out})\}$ depicted in Fig. 5. Since $Args = \mathtt{in}(Lab_{part}) \backslash \mathtt{in}(Lab) = \{f\} \neq \emptyset$, line 4 of Alg.2 returns $\{f\}$ as the error: f should not be labelled $\mathtt{in}$ as it is attacked by an argument which should be labelled $\mathtt{in}$, namely argument a (see Fig.4).

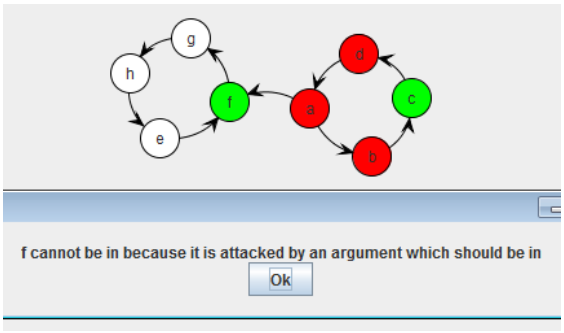

Figure 4. The predefined $AF7$ with the incorrect partial labelling Lab_{part} (see Example 2) and an error message computed by line 4 of Alg.2.

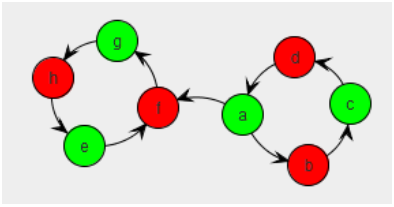

Figure 5. An out-maximal complete labelling Lab with respect to Lab_{part} from Example 2.

VII. Related Work

Various algorithms using the labelling of arguments for the computation of AA semantics can be found in the literature [14], [13], [15], [16], most of which focus on the preferred instead of the complete semantics. All these algorithms follow the idea that an argument should be labelled $\mathtt{out}$ if it is attacked by some $\mathtt{in}$ argument, and $\mathtt{in}$ if all its attackers are labelled $\mathtt{out}$, but none of them uses our new method of in-self-defending arguments for extending the set of $\mathtt{in}$ arguments (see Alg.1).

Verheij [14] outlines an algorithm for computing credulous acceptance with respect to the preferred semantics, i.e. whether an argument is labelled $\mathtt{in}$ in at least one preferred labelling. Starting from the argument in question being $\mathtt{in}$ and all other arguments $\mathtt{undec}$, the $\mathtt{undec}$ arguments are iteratively relabeled as $\mathtt{out}$ if they are attacking an $\mathtt{in}$ argument, or as $\mathtt{in}$ if this maintains conflict-freeness of the $\mathtt{in}$ arguments. An implementation of this approach in Delphi is available online[6]. Modgil and Caminada [13] present an algorithm for computing preferred labellings by starting from all arguments being $\mathtt{in}$ and then relabelling them as $\mathtt{out}$ or $\mathtt{undec}$ if necessary. In contrast to Alg.1, the algorithms in [14] and [13] operate on total labellings.

In [15] an algorithm for finding all preferred labellings using five labels is proposed: $\mathtt{in}$, $\mathtt{out}$, $\mathtt{undec}$, $\mathtt{must\_out}$, and $\mathtt{blank}$, where $\mathtt{blank}$ denotes unlabelled arguments and $\mathtt{must\_out}$ arguments attacking $\mathtt{in}$ arguments. Similar to Alg.1, the algorithm starts with all arguments being

[6]http://www.ai.rug.nl/~verheij/comparg/

`blank`, but then chooses one unlabelled argument randomly in every iteration to label it `in` or `undec`. If labelled `in`, all attacking arguments are labelled `must_out` and all attacked arguments `out`. This procedure is repeated until all arguments are labelled `in`, `out`, or `undec`. ArgTools[7] implements this algorithm in C++.

ArguLab[8] [17] is another implementation using labellings, but no information about the underlying algorithm is available. Even considering implementations of AA semantics which do not use labellings for the computation, like ASPARTIX [9] or Dungine [18], none of them is written in Prolog, making ARGTEACH the first documented Prolog implementation of AA semantics.

VIII. CONCLUSION

We presented ARGTEACH, a software for teaching Abstract Argumentation (AA) semantics in terms of labellings. The main teaching features of ARGTEACH are hints about which argument to label next and error-checking of the current partial labelling. In addition to being the first implementation of AA semantics designed for teaching support, ARGTEACH is also the first documented implementation written in Prolog.

Different from existing implementations of AA semantics, ARGTEACH operates on partial labellings, where some arguments are left unlabelled during the labelling process. Furthermore, a novel method for the step-by-step computation of complete labellings is used, involving sets of "inself-defending" arguments, i.e. non-empty sets of unlabelled arguments which are able to defend themselves together with the current `in` arguments.

Since admissible labellings [10] are not based on complete labellings, they are currently not supported by ARGTEACH, but will be integrated in future work. Given that correct partial labellings and admissible labellings are defined in a similar way, computing admissible labellings in ARGTEACH will be reduced to finding correct partial labellings. Furthermore, we plan to conduct experiments with students to compare their understanding of AA semantics with and without as well as before and after using ARGTEACH.

REFERENCES

[1] P. M. Dung, "On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games," *AI*, vol. 77, no. 2, pp. 321–357, 1995.

[2] P. Baroni, M. Caminada, and M. Giacomin, "An introduction to argumentation semantics," *The Knowledge Engineering Review*, vol. 26, pp. 365–410, 2011.

[3] P. Baroni and M. Giacomin, "On principle-based evaluation of extension-based argumentation semantics," *AI*, vol. 171, no. 10–15, pp. 675–700, 2007.

[4] M. Caminada, W. A. Carnielli, and P. E. Dunne, "Semi-stable semantics," *Journal of Logic and Computation*, vol. 22, no. 5, pp. 1207–1254, 2011.

[5] P. Torroni, M. Gavanelli, and F. Chesani, "Argumentation in the semantic web," *Intelligent Systems, IEEE*, vol. 22, no. 6, pp. 66–74, 2007.

[6] M. Caminada and M. Podlaszewski, "Grounded semantics as persuasion dialogue," in *COMMA'12*, 2012, pp. 478–485.

[7] T. J. Bench-Capon, "Representation of case law as an argumentation framework," in *JURIX'02*, 2002, pp. 103–112.

[8] C. Reed and G. Rowe, "Araucaria: Software for argument analysis, diagramming and representation," *International Journal on AI Tools*, vol. 13, no. 4, pp. 961–979, 2004.

[9] U. Egly, S. A. Gaggl, and S. Woltran, "Aspartix: Implementing argumentation frameworks using answer-set programming," in *ICLP'08*, 2008, pp. 734–738.

[10] M. Caminada and D. Gabbay, "A logical account of formal argumentation," *Studia Logica*, vol. 93, pp. 109–145, 2009.

[11] B. P. Woolf, *Building Intelligent Interactive Tutors: Student-centered Strategies for Revolutionizing e-Learning*. Morgan Kaufmann Publishers Inc., 2007.

[12] G. Wiggins, *Educative Assessment. Designing Assessments To Inform and Improve Student Performance*. Jossey-Bass Publishers, 1998.

[13] S. Modgil and M. Caminada, "Proof theories and algorithms for abstract argumentation frameworks," in *Argumentation in AI*. Springer US, 2009, pp. 105–129.

[14] B. Verheij, "A labeling approach to the computation of credulous acceptance in argumentation," in *IJCAI'07*, 2007, pp. 623–628.

[15] S. Nofal, K. Atkinson, and P. E. Dunne, "Algorithms for decision problems in argument systems under preferred semantics," *AI*, vol. 207, pp. 23–51, 2014.

[16] M. Caminada, "An algorithm for computing semi-stable semantics," in *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. Springer Berlin Heidelberg, 2007, vol. 4724, pp. 222–234.

[17] M. Podlaszewski, M. Caminada, and G. Pigozzi, "An implementation of basic argumentation components," in *AAMAS'11*, 2011, pp. 1307–1308.

[18] M. South, G. Vreeswijk, and J. Fox, "Dungine: A java dung reasoner," in *COMMA'08*, 2008, pp. 360–368.

[7] http://sourceforge.net/projects/argtools/

[8] http://heen.webfactional.com/