A Notation for Declaratively Describing Arguments

Simon Wells[1,*], Mark Snaith[2]

[1]Edinburgh Napier University, 10 Colinton Road, Edinburgh, EH10 5DT, Scotland, UK
[2]Robert Gordon University, Garthdee House, Aberdeen, AB10 7QB, Scotland, UK

Abstract
We introduce the Simple Argument Description Notation (SADN), a declarative language for describing arguments. SADN is built upon the existing, document-oriented, SADFace argument description format and is used to interactively and decalaratively construct argument descriptions. A worked illustrated example is presented in which an argument model is constructed through the application of SADN.

Keywords
Argument Description, Notation, Declarative Programming,

1. Introduction

The Simple Argument Description Notation (SADN[1]) is a language for describing arguments in an incremental, line-oriented declarative fashion. SADN is built upon the Simple Argument Description Format (SADFace) [1] which provides a core ontology of argumentative concepts and an implemented library for constructing and manipulation SADFace documents using either Python3 or Javascript.

The core idea of SADN is that a SADFace document is a dynamic model of an argument and that the user makes line oriented declarations such that each declaration modifies the argument description, leaving a complete, coherent and consistent SADFace model at each stage. This builds upon prior work on declarative dialogue description languages [2]. The user might start with an empty model, or load an existing model for further manipulation and then execute SADN declarations where each declaration is a command in the Read-Evaluate-Print Loop (REPL) and each line of the REPL represents a distinct, indexed, temporal timestep in the life of the argument model. A key conceptual difference therefore between SADFACE and SADN is that whereas SADFace is a single, multi-line document, in SADN the SADFace document is treated as a dynamic model that exists wholly and fully formed at each time step as indexed by REPL lines. SADN and SADFace can therefore be considered as isomorphic modes of interaction with the same underlying argumentative model.

One motivation for this work is to better capture the dynamic process that relates dialogical dynamics to a argumentative structure. After any given utterance within a dialogue, there exists a corresponding argumentative structure that models the state of the arguments, made by the participants, for and against the issue that is at stake. This initial version of SADN focusses solely upon capturing the dynamic construction aspect of the argument model and leaves the next stage of capturing dialogue dynamics to future work.

In the remainder of this paper we begin by briefly discussing related work, cover some background material, in the form of a brief introduction to the SADFace format and library implementation upon which SADN builds, then introduce an annotated presentation of the SADN commands. We present an example of a SADN session and some discussion of supporting tooling and applications before concluding with a discussion of future directions.

CMNA'25: The Twenty-Fifth International Workshop on Computational Models of Natural Argument, December 12, 2025, Online
*Corresponding author.
s.wells@napier.ac.uk (S. Wells); m.snaith@rgu.ac.uk (M. Snaith)
https://www.simonwells.org (S. Wells); https://www3.rgu.ac.uk/dmstaff/snaith-mark/ (M. Snaith)
0000-0003-4512-7868 (S. Wells); 0000-0001-9979-9374 (M. Snaith)

[1]pronounced "*sadden*"

2. Related Work

Two archetypal prior approaches to declarative argument descriptions include DeLP [3] and ASPIC+ [4]. However neither is a general purpose language for describing arguments, each of which is associated with a particular, opinionated approach to, defeasible logic programming in the case of DeLP, and structured argument evaluation in the case of ASPIC+. With SADN we seek to develop a general purpose notation for describing arguments which can subsequently interoperate with other downstream and upstream tools, including, but not limited to DeLP [3], ASPIC+ [4], AIF [5], TOAST [6], and others.

3. The Simple Argument Description Format (SADFace)

SADN builds directly upon the core concepts defined within SADFace, as an argument description format, whilst also utilising the argumentative model built into the SADFace libary implementation. Therefore it is sensible to briefly describe those features of SADFace most pertinent to SADN.

SADFace is a Javascript Object Notation (JSON) [7] based document format for describing analysed argument structure and associated meta-data. The aim of this work is to build argument tools for the web. That is simple, flexible, extensible tools that fit into existing web-workflows and tool-chains. A developer should be able to adopt SADFace and start describing arguments or using those descriptions, really easily. Parsing a SADFace document into a Javascript application should not require any special tools, it is just JSON. The structure of the format has been designed to align with a straightforward model of argument structure, defined in such a way as to align with most of the intuitions that an everyday understanding of argument will include, whilst still supporting more advanced features.

Arguments are made up from statements (or strings if you're a programmer) that capture either a "claim" or supporting information for that claim. In SADFace these statements are called "Argument Atoms", or just "atoms". A simple argument is a collection of such atoms that are linked together. For a simple, single, argument we generally say that one of the atoms is a conclusion, and the other atoms are premises supporting that conclusion. The exact way that premises support a conclusion is defined by an argumentation scheme. An argumentation scheme captures a stereotypical pattern of argumentative reasoning or inference, so different arguments can be categorised into different types. A rough rule of thumb within SADFace is to consider every individual argument is being an instance of an argumentation scheme.

The underlying model of SADFace is the Mathematical Graph [8], a collection of nodes connected together by edges. More spectifically, SADFace uses a directed graph. In this case the edges have a direction, they claim a relationship that works in a particular direction between any pair of nodes that are connected by a given edge. A SADFace document contains a set of nodes and a set of edges. Edges are quite simple, they have an ID so that the instance of the edge can be uniquely identified from all other edges, and they also have a source and a target. The source and target in an edge are both IDs of nodes, i.e. the ID of the node that this edge goes from, the source ID, and the ID of the node that this edge goes to, the target ID. Nodes are slightly more complicated, currently there are two types of node, atom nodes and scheme nodes. Atoms and schemes are linked together using the aforementioned edges. There are also some restrictions upon which types of node can be lined directly to each other:

Atoms cannot be linked directly to each other. They must be linked via a Scheme node. This is in line with the underlying model of the Argument Interchange Format (AIF) [5] which restricts Information or i-nodes, the equivalent of our Atoms, from being linked directly to each other. This is an important restriction as it means that for every link between a pair of atoms, for the link to be valid, it must define a known argumentative relationship.

Figure 1 shows a fragment of JSON that presents an abbreviated illustration of an argument described in SADFace. Note that only one example of each element is included and the remainder of the full analysis has been elided for brevity and replaced with '…'.

This description yields a visualisation, using the MonkeyPuzzle argument diagramming layout

```
{
  "edges": [
    {
      "id": "b00535db-25b8-45b4-99e9-2955fa0ba54c",
      "source_id": "9cfd99f2-2cd3-4fba-8580-7e76add66e8b",
      "target_id": "df5d8a22-bf5d-4da3-b902-ccd45a33f5c2"
    },
    ...
  ],
  "metadata": {
    "core": {
      "analyst_email": "s.wells@napier.ac.uk",
      "analyst_name": "Simon Wells",
      "created": "2025-02-23T02:27:36",
      "edited": "2025-10-08T16:52:32",
      "id": "94a975db-25ae-4d25-93cc-1c07c932e2f9",
      "version": "0.5.3.2"
    }
  },
  "nodes": [
    {
      "id": "9cfd99f2-2cd3-4fba-8580-7e76add66e8b",
      "metadata": {},
      "sources": [],
      "text": "If you are going to die then you should treasure every
          moment",
      "type": "atom"
    },
    {
      "id": "df5d8a22-bf5d-4da3-b902-ccd45a33f5c2",
      "name": "Support",
      "type": "scheme"
    },
    ...
  ],
  "resources": []
}
```

Figure 1: An illustrative SADFace document that shows the four main organisational blocks of the SADFace model, including the edge list, node list, resource list, and metadata block.

method [1], as show inf Figure 2. SADFace has a canonical Python implementation[2] which is free software available through the Python Package Index (PyPi[3]) and released under the Gnu General Public License (GPL) v.3.0 as a part of the wider Open Argumentation Platform[4] [9].

4. The Simple Argument Description Notation (SADN)

SADN is designed primarily to be used in a terminal or REPL, or else piped between command line applications. The result of any given SADN expression is an argument structure that is a syntactically valid[5] SADFace model.

[2]$ pip install sadface
[3]https://pypi.org/
[4]http://www.openargumentation.org
[5]semantic validation is left to future work.

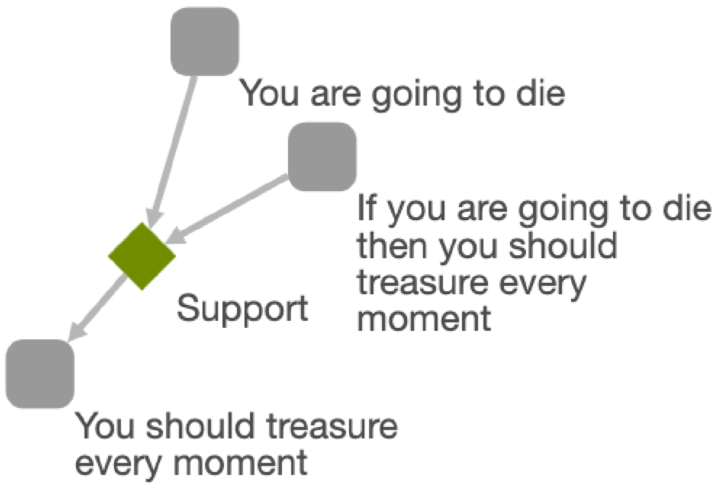
You are going to die
If you are going to die then you should treasure every moment
Support
You should treasure every moment

Figure 2: A visualisation of the argument described in the SADFace document shown in Figure 1. Atoms are depicted using grey boxes, generic 'support' schemes using green boxes, and arrows indicate the direction of the relationship between the nodes at each end.

4.1. SADN Commands

The following is an annotated list of the core SADN commands for declarative SADFace argument model construction and manipulation. Each command is depicted in bold and the annotations in regular text. Some elements are depicted using the following form **$UPPERCASE** to indicate variables or elements that are not SADN commands. Such elements includes identifiers, and text string values associated with SADFace keys. Additionally, the vertical bar '|' is used to indictate a choice between alternatives, e.g.'a | b', meaning 'a or b'.

atom(id:$ID, text:$TEXT) – Add a new atom node. $CONTENT is an optional quoted string containing the textual content of the atom. If the atom command is executed with an existing $ID and a new text $TEXT then this will cause the existing atom node's text value to be updated with the new $TEXT string.

scheme(id:$ID, name:$TEXT) – Add a new scheme node. $TEXT is an optional quoted string containing the name of the associated argumentation scheme. If no $TEXT is supplied then a generic "support" name will be used. If the scheme command is executed with an existing $ID and a new name $TEXT then the existing scheme node will be updated with the new name.

Commentary: The atom and scheme commands are grouped together because they have the same basic form and have similar effects. Each command can be run without arguments to yield, respectively, the introduction of a single SADFace atom or scheme node with a generic name/content and automatically generated ID into the SADFace model.

support(scheme:$ID, conclusion:$ID, premise:$ID, ...) – Create a new supportive relationship between the existing nodes identified by the conclusion and premise IDs. The *support* command allows an arbitrary number of premise IDs to be supplied. If no scheme ID is supplied then a new generic "support" node is created .

attack(source:$ID, target:$ID) – Create a new attack relationship between the supplied nodes identified by the source and target $IDs. Attack is a uni-directional conflict from one argument that is directed towards another argument. A new conflict node of type *attack* is created as a result.

disagree($ID, $ID, ...) – Create a new disagreement relationship between the nodes identified by the supplied $IDs. Disagree is a bi-directional conflict between two or more arguments A new conflict node of type *disagreement* is created as a result.

Commentary: The support, attack, and disagree commands are grouped together because they work similarly, used to create relationships between nodes, but have distinct individual semantic interpretations in terms of how they affect the SADFace model.

argument(scheme:$TEXT, conclusion:$ID | text:$CONTENT, premise:$ID | text:$CONTENT, ...) – Builds a complete argument structure in a single command. If no name $TEXT is supplied then a generic 'support' label will be added to the scheme. The remainder of the arguments are a comma separated list of arbitrary length of either $IDs referring to pre-existing nodes or double-quoted $TEXT strings which cause new atoms to be created. The first ID supplied will be interpreted as the concluding node, all other IDs will be interpreted as premises.

Commentary: The argument command is syntactic sugar to enable an entire argument structure to be constructed in one declaration and in a fairly ad-hoc manner using a mixture of IDs for pre-existing atoms and content for atoms to be created during the execution of this command. The usefulness of the argument() command for rapidly introducing entire argument structures comes at the expense of some additional complexity.

remove(id:$ID, ...) – Remove the element identified by $ID from the model. This also removes any dependent elements within the SADFace model such as edges if the removed element is either the source or target of the edge.The remove operation is idempotent and repeated use on the same $ID will make no further change to the model. If no $ID is supplied the remove has no effect on the model.

Commentary: The remove command is unique in that it is the only command that will remove specified elements from the SADFace model

metadata(id:$ID, namespace:$TEXT, $KEY:$TEXT) – Used to add a metadata entry to the identified namespace of the SADFace element identified by $ID. Elements that can support metadata blocks are the model's global metadata block, individual nodes, and individual resources. If the $KEY already exists then the $TEXT updates itand if the $KEY doesn't exist then that key is created.

Commentary: Namespaced metadata at defined places in the model is a key to SADFace's claim to extensibility. The metadata command enables metdata to be captured as required during declarative SADN REPL sessions.

atoms - Lists the atom nodes in the current SADFace model.

schemes - Lists the scheme nodes in the current SADFace model.

conflicts - Lists the conflict nodes, in the current SADFace model which includes both attacks and disagreements.

edges - Lists the atom nodes in the current SADFace model.

Commentary: These are used primarily to see what nodes are in the current model and to access the mapping from UUID to local alias. Because the elements that make up a SADFace model can be spread over an extended SADN REPL session and can be difficult to keep track of, it is useful to be able to access, list, group, and display the elements of the model so that they can be more easily manipulated. These commands work in concert with the print and prettyprint commands to support interactive manipulation of the SADFace model.

export(type:$TYPE) – SADFace supports exporting the SADFace model into a range of other text-based forms, for example, Dot and Cytoscape. The export keyword converts the SADFace model into the nominated model and displays it in the REPL.

save(type:$TYPE, name:$PATHNAME) – Saves the SADFace model to a file on disk. Save supports all of the textual formats valid in the export keyword but adds binary formts that can't be usefully displayed in the REPL including jpg, png, and svg formats.

load(name:$PATHNAME) – Loads the SADFace document from the supplied pathname.

clear – Return the current situation to an empty default SADFace model.

print – Display the SADFace model as a single compact string.

prettyprint – Display the SADFace model in a pretty printed JSON form using tabs and newlines.

help – Display a list of available commands.

Commentary: This final grouping of commands are all utility functions for manipulating, importing, exporting, and working with SADN but don't directly modify the SADFace model.

Notice that some commands are *complex* and take one or more arguments which are embraced in parenthesis whereas other *simple* commands include neither arguments nor parenthesis. Of the commands that take arguments, notice that the argument lists are explicitly structured as comma separated lists of key:value pairs.

In any command that consumes an $ID (except for the remove command, if no $ID is supplied then a new UUID and local UUID Alias will be created that are unique to the SADFace elements being created. If the same command is called without specified $IDs then that command will complete each time with new elements created containing generated IDs.

Where commands can support more complex relationship structures, such as in the support, attack, and disagree relationships, the first $ID is interpreted as the *head* element, and any remaining $ID elements are interpreted as the *tail* element(s). For example, support(id:a1, id:a2, id:a3) yields an argument where atom a1 is the conclusion, and the remaining atoms a2 and a3 are premises, all of which are linked via a 'support' scheme node.

4.2. UUID Aliasing

One challenging user experience issue within SADN is the use of long, and quite unwieldy, universally unique identifiers (UUIDs) which are core to how SADFace creates globally unique argument instances. See Figure 1 for an illustration of a range of UUIDs associated with the various "id" keys in SADFace. Manipulating UUIDs within the command line involves either error prone typing or else copy-pasting. SADN introduces transparent local alisasing as syntactic sugar to make it easer to use SADN without manipulating otherwise unwieldy UUIDS in the terminal and thus to improve the user experience. UUIDs are replaced locally with short monotonically increasing unique identifiers of the basic form **Letter Code-Numeric ID**. For example, an Atom, having textual content, associated metadata, and a unique ID might be depicted as follows in SADFace:

```
{
   "id": "ae3f0c7f-9f69-4cab-9db3-3b9c46f56e09",
   "metadata": {},
   "sources": [],
   "text": "The 'Hang Back' campaign video should be withdrawn.",
   "type": "atom"
}
```

But when instantiated in SADN, the atom identified by the UUID "ae3f0c7f-9f69-4cab-9db3-3b9c46f56e09" might be referred to using the term **A-1** for Atom 1 within that specific SADN session. Each SADFace element is treated similarly, assigned a letter code based upon type and a numeric identifier based upon when the element was expressed during the current SADN session. Internally SADN session IDs are mapped to original UUIDs using a lookup table. If a SADFace document is read into a SADN session then the original UUIDs are maintained in the lookup table and SADN sessions IDs are generated incrementally as the document is processed. If a SADN session starts with an empty document, then session IDs are generated incrementally as needed as the session progresses, UUIDs are also generated for each element so that all SADFace models, generated through SADN, can be serialised to universally valid SADFace documents.

4.3. Tooling

A limited prototype of SADN is included in the current SADFace release. To access this, you will need a working Python 3 installation. SADFace can then be installed using pip, i.e. *pip install sadface*[6]. From there, the SADFace REPL can be initiated using *python3 -m sadface -i*[7]. A SADFace configuration file can be passed into the invocation of the SADN REPL which will cause the new SADFace model to be built using configuration information, such as analyst email and name, from the file. This is achieved using the following command *python -m sadface -c test.cfg -i* where the configuration is stored in the test.cfg file.

5. Worked Illustrated Example

In this example, we'll recreate the example argument from the MonkeyPuzzle demo. We'll show the commands in the REPL, and where appropriate, the SADFace model and MonkeyPuzzle visualisation. Upon starting the SADFace REPL without a configuration file or any other inputs you will be presented with the following prompt at which you can type SADN commands:

```
The SADFace REPL. Type 'help' or 'help <command>' for assistance
>
```

The associated SADFace model is empty, containing neither nodes nor edges or resources and the core metadate will contain default information for the analyst name and email address keys, current timetstamps for the created and edited keys, a universally unique identifier for the ID key, and the semantic version number for the version of SADFace associated with the model. By executing the *pp* command, the abbreviated version of the prettyprint command from the list introduced in section 4.1, the current model will be displayed as follows:

```
The SADFace REPL. Type 'help' or 'help <command>' for assistance
> pp
{
    "edges": [],
    "metadata": {
        "core": {
        "analyst_email": "Default Analyst Email",
        "analyst_name": "Default Analsyst Name",
        "created": "2025-10-29T17:53:05",
        "edited": "2025-10-29T17:53:30",
        "id": "4ab333ea-a003-4631-afa7-5f20806a8497",
        "version": "0.5.3.2"
    }
    },
    "nodes": [],
    "resources": []
}
>
```

We can now add some nodes and edges. to the model. Let's start by adding the nodes for the initial sub-argument, then we'll use the print command and the list("atoms") commands to show the effect on the model:

[6]It is best practise to install the SADFace tooling into a virtual environment

[7]NB. The invocation of Python might differ across platforms and installations.

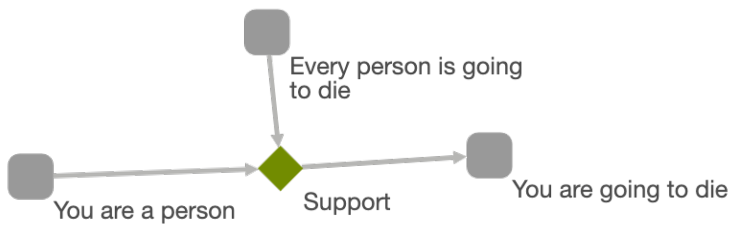

Figure 3: The first sub-argument in the MonkeyPuzzle demonstration argument diagram. This is the result after executing three *atom()* commands and a single *support()* command.

```
> atom(text:"Every person is going to die")
> atom(text:"You are a person")
> atom(text:"You are going to die")
> print
{"edges":[],"metadata":{"core":{"analyst_email":"Default Analyst Email","analyst_name":"
    Default Analsyst Name","created":"2025-10-29T17:53:05","edited":"2025-10-29T17
    :53:30","id":"4ab333ea-a003-4631-afa7-5f20806a8497","version":"0.5.3.2"}},"nodes
    ":[{"id":"a03ea3f2-70f2-42ed-962c-f15367e79de7","type":"atom","text":"Every person
    is going to die","sources":[],"metadata":{}},{"id":"6e73e177-4e62-45e4-bd0d-62
    d72891c339","type":"atom","text":"You are a person","sources":[],"metadata":{}},{"id
    ":"bba487d5-029a-4a1b-ae79-ad69c769382a","type":"atom","text":"You are going to die
    ","sources":[],"metadata":{}}],"resources":[]}
>atoms
["a1":{"id":"a03ea3f2-70f2-42ed-962c-f15367e79de7", "text":"Every person is going to die
    "}, "a2":{"id":"6e73e177-4e62-45e4-bd0d-62d72891c339", "text":"You are a person"}, "
    a3":{"id":"bba487d5-029a-4a1b-ae79-ad69c769382a", "text":"You are going to die"}]
```

Now we'll add the edges to join the atoms together into an argumentative structure. We'll use the *support* command to do this. We won't specify a scheme ID as we haven't created a scheme node yet. Instead we'll accept the scheme node generation as a side effect of creating the support structure. So we'll only identify the conclusion and premises, using the local atom IDs output from the list command above and applying *conclusion* and *premise* labels. We'll then use the *schemes* command to see the list of scheme nodes, now that one has been created, and finish by diaplying the current model using *print* once more:

```
> support(conclusion:"a3", premise:"a1", premise:"a2")
>schemes
["s1":{"id":"b3b0e13a-8b8b-4e2a-9544-5ab401e58999", "name":"Support"}]
> print
{"edges":[{"id":"81d94ca5-4cb1-4c4d-8486-708d45b45c4e","source_id":"a03ea3f2-70f2-42ed
    -962c-f15367e79de7","target_id":"b3b0e13a-8b8b-4e2a-9544-5ab401e58999"},{"id":"233
    af9c1-ba7e-4967-9c32-3025ac65ee67","source_id":"b3b0e13a-8b8b-4e2a-9544-5ab401e58999
    ","target_id":"bba487d5-029a-4a1b-ae79-ad69c769382a"},{"id":"75c9ef27-7ba4-4cbf-8fde
    -e957698f472b","source_id":"6e73e177-4e62-45e4-bd0d-62d72891c339","target_id":"
    b3b0e13a-8b8b-4e2a-9544-5ab401e58999"}],"metadata":{"core":{"analyst_email":"Default
     Analyst Email","analyst_name":"Default Analsyst Name","created":"2025-10-29T17
    :53:05","edited":"2025-10-29T18:02:25","id":"4ab333ea-a003-4631-afa7-5f20806a8497","
    version":"0.5.3.2"}},"nodes":[{"id":"a03ea3f2-70f2-42ed-962c-f15367e79de7","type":"
    atom","text":"Every person is going to die","sources":[],"metadata":{}},{"id":"6
    e73e177-4e62-45e4-bd0d-62d72891c339","type":"atom","text":"You are a person","
    sources":[],"metadata":{}},{"id":"bba487d5-029a-4a1b-ae79-ad69c769382a","type":"atom
    ","text":"You are going to die","sources":[],"metadata":{}},{"id":"b3b0e13a-8b8b-4
    e2a-9544-5ab401e58999","type":"scheme","name":"Support"}],"resources":[]}
```

Now is an opportune moment to view the visualisation of the argument structure constructed so far. We'll use MonkeyPuzzle to do this using the SADFace model from the REPL. The result is shown in Figure 3

Finally we'll complete the argument map from the MonkeyPuzzle demonstration but this time we'll use the *argument()* command to combine a number of simpler commands into a single declaration. This combines supplying a mixture of local IDs from existing atoms, and strings to create new nodes or reuse existing nodes, as required within the same command to build an argument structure in a single declaration. Whilst such syntactic sugar is not necessary, and the same effect can be built from primitive SADN commands, this approach does reduce the number of commands necessary to achive the same effect.

```
> argument(conclusion: "You should treasure every moment", premise: "a3", premise: "If
    you are going to die then you should treasure every moment")
> atoms
["a1":{"id":"a03ea3f2-70f2-42ed-962c-f15367e79de7", "text":"Every person is going to die
    "}, "a2":{"id":"6e73e177-4e62-45e4-bd0d-62d72891c339", "text":"You are a person"}, "
    a3":{"id":"bba487d5-029a-4a1b-ae79-ad69c769382a", "text":"You are going to die"}, "
    a4":{"id":"3b666117-d50c-4de8-b2ed-29f6ae98f10a", "text":"You should treasure every
    moment"}, "a5":{"id":"df8e97d1-b2d6-4768-bd82-ea91a8820313", "text":"If you are
    going to die then you should treasure every moment"}]
> schemes
["s1":{"id":"b3b0e13a-8b8b-4e2a-9544-5ab401e58999", "name":"Support"}, "s2":{"id":"42
    ba2acc-d8d2-4d8f-ae93-c7f8c383b4bb", "name":"Support"}]
> print
{"edges":[{"id":"81d94ca5-4cb1-4c4d-8486-708d45b45c4e","source_id":"a03ea3f2-70f2-42ed
    -962c-f15367e79de7","target_id":"b3b0e13a-8b8b-4e2a-9544-5ab401e58999"},{"id":"233
    af9c1-ba7e-4967-9c32-3025ac65ee67","source_id":"b3b0e13a-8b8b-4e2a-9544-5ab401e58999
    ","target_id":"bba487d5-029a-4a1b-ae79-ad69c769382a"},{"id":"75c9ef27-7ba4-4cbf-8fde
    -e957698f472b","source_id":"6e73e177-4e62-45e4-bd0d-62d72891c339","target_id":"
    b3b0e13a-8b8b-4e2a-9544-5ab401e58999"},{"id":"e21ddbcc-ce50-4ce9-839e-5f8f5e2d5af4
    ","source_id":"df8e97d1-b2d6-4768-bd82-ea91a8820313","target_id":"42ba2acc-d8d2-4d8f
    -ae93-c7f8c383b4bb"},{"id":"adbc1971-5334-4daa-86bb-d7d706a3b8cc","source_id":"42
    ba2acc-d8d2-4d8f-ae93-c7f8c383b4bb","target_id":"3b666117-d50c-4de8-b2ed-29
    f6ae98f10a"},{"id":"00695abb-e8aa-4f08-9af8-7fca55d53270","source_id":"bba487d5-029a
    -4a1b-ae79-ad69c769382a","target_id":"42ba2acc-d8d2-4d8f-ae93-c7f8c383b4bb"}],"
    metadata":{"core":{"analyst_email":"Default Analyst Email","analyst_name":"Default
    Analsyst Name","created":"2025-10-29T17:53:05","edited":"2025-10-29T18:50:13","id
    ":"4ab333ea-a003-4631-afa7-5f20806a8497","version":"0.5.3.2"}},"nodes":[{"id":"
    a03ea3f2-70f2-42ed-962c-f15367e79de7","type":"atom","text":"Every person is going to
     die","sources":[],"metadata":{}},{"id":"6e73e177-4e62-45e4-bd0d-62d72891c339","type
    ":"atom","text":"You are a person","sources":[],"metadata":{}},{"id":"bba487d5-029a
    -4a1b-ae79-ad69c769382a","type":"atom","text":"You are going to die","sources":[],"
    metadata":{}},{"id":"b3b0e13a-8b8b-4e2a-9544-5ab401e58999","type":"scheme","name":"
    Support"},{"id":"df8e97d1-b2d6-4768-bd82-ea91a8820313","type":"atom","text":"If you
    are going to die then you should treasure every moment","sources":[],"metadata
    ":{}},{"id":"3b666117-d50c-4de8-b2ed-29f6ae98f10a","type":"atom","text":"You should
    treasure every moment","sources":[],"metadata":{}},{"id":"42ba2acc-d8d2-4d8f-ae93-
    c7f8c383b4bb","type":"scheme","name":"Support"}],"resources":[]}
```

The final Monkeypuzzle visualisation of the argument constructed in the SADN REPL can be seen in Figure 4

6. Applications

There are a number of immediate applications for SADN. The primary application, in line with the earlier example, is as an interactive declarative language, with supporting utilities, for constructing argument models without needing to resort to manual argument diagramming software like MonkeyPuzzle. This approach has been demonstrated in the worked example in section 5. MonkeyPuzzle and SADFace are not the only target applications however, and the core commands could easily be reused to work with other argument models and descriptions. One easy route towards this is through extending the export

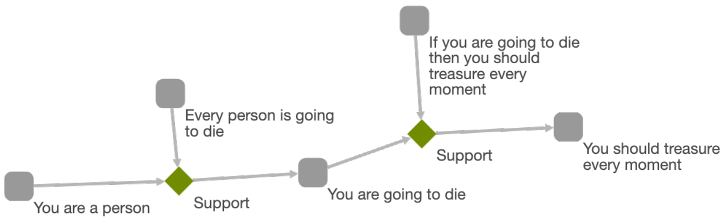
If you are going to die then you should treasure every moment
Every person is going to die
Support
You should treasure every moment
You are going to die
You are a person
Support

Figure 4: The full reconstruction of the MonkeyPuzzle demonstration argument diagram. This is the result after executing three *atom()* commands and a single *support()* command to create the sub-argument, then the addition of a single *argument* command to build the concluding argument.

functionality of SADFace to account for new models as they become available. A second application of SADN is within an interactive argument evaluation context. In this application, SADN commands are used to build a model that is then piped through additional tools to evaluate the argument on the fly, for example, piping the state of the SADFace model from the SADN REPL through to a tool like TOAST in order to evaluate the constructed model under the ASPIC+ theory. On design goal of SADN was to produce a declarative language that could be used interactively when working with arguments, that covers the core concepts of argumentative practise. This is similar to the goal of SADFace to cover those same concepts from a document-oriented structured graph, perspective. This contrasts with the goals of AIF, for example, which seeks to be a comprehensive upper ontology of argumentative concepts, an approach that can bring additional complexity and reduce clarity for end-users. One additional application that we are investigating is the use of SADN when providing structured argumentative information to Large Language Models (LLMs) [10] in a prompt-style interface [11, 12]. Many argument description formats are in graph-based file formats, expressed in JSON or XML, and whilst LLMs have been gaining capabilities in parsing structure from such graphs, the underlying arugmentative relationships captured therein have proven elusive. The main interface to LLMs is textual, and interactive. Users prompt the LLM to provide hopefully relevant responses and over multiple interactions, previous results can affect future results, leading to a dialogue-like structure.

7. Discussion & Future Work

There are a number of directions for future work. Firstly, refining the user experience of the SADN notation by implementing more syntactic sugar and shortcuts to make input and manipulation simpler and faster for experienced users. Secondly, improving the user experience within the SADFace REPL by using suggested auto-complete data from existing variables and elements within the current model. Thirdly, providing additional queues within the REPL using colour to differentiate, i.e. keywords from variables and elements. Fourthly, extending the notation to support construction of dialogues along with the necessary machinery to link arguments with their associated dialogical elements.

In conclusion, SADN enables us to efficiently build an argument description, declaratively, interactively, and incrementaly, in an efficient manner using a notation built upon the existing SADFace argument description model.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] S. Wells, D. J., Monkeypuzzle: Towards next generation, free & open-source, argument analysis tools, in: Proceedings of the 17th International Workshop on Computational Models of Natural Argument (CMNA17), 2017.

[2] S. Wells, D. J., Towards a declarative approach to constructing dialogue games, in: Proceedings of the 21st International Workshop on Computational Models of Natural Argument (CMNA21), 2021, pp. 9–18.

[3] A. J. Garcia, G. R. Simari, Defeasible logic programming: Delp-servers, contextual queries, and explanations for answers, Argument & Computation 5 (2014) 63–88.

[4] S. Modgil, H. Prakken, The aspic+ framework for structured argumentation: a tutorial, Argument & Computation 5 (2014) 31–62.

[5] C. Chesnevar, J. McGinnis, S. Modgil, I. Rahwan, C. Reed, G. Simari, M. South, G. Vreeswijk, S. Willmott, Towards an argument interchange format, Knowledge Engineering Review 21 (2006) 293–316.

[6] M. Snaith, C. Reed, Toast: online aspic+ implementation, in: Proceedings of the Fourth International Conference on Computational Models of Argument, IOS Press, 2012, pp. 509–510.

[7] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, D. Vrgoč, Foundations of json schema, in: Proceedings of the 25th International Conference on World Wide Web, International World Wide Web Conferences Steering Committee, 2016, pp. 263–273.

[8] R. Diestel, Graph Theory, Springer Berlin, Heidelberg, 2025.

[9] S. Wells, The open argumentation platform (OAPL), in: Proceedings of Computational Models of Argument. (COMMA 2020), Frontiers in Artifical Intelligence, IOS Press, 2020, pp. 465–476.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, Curran Associates Inc., Red Hook, NY, USA, 2017, p. 6000–6010.

[11] S. Wells, M. Snaith, On the role of dialogue models in the age of large language models, in: Proceeding of the the 23rd International Workshop on Computational Models of Natural Argument, CEUR, 2023.

[12] M. Snaith, S. Wells, Retrieval augmented generation for immersive formal dialogue, in: Proceedings of the Fifth European Conference on Argumentation, College Publications, to Appear.