H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable D4.1

Constraint Definition Language

Version: 1.0

Publication Date: 31-December-2019

Deliverable Card

Deliverable	D4.1
Title:	Constraint Definition Language
Editor(s):	Mark Law (IMP)
Contributor(s):	Mark Law (IMP), Alessandra Russo (IMP)
Reviewers:	Michael Wurster (UST), Mike Long (PRQ)
Type:	Report
Version:	1.0
Date:	31-December-2019
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables
Copyright:	RADON consortium

The RADON project partners

IMP	IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE
TJD	STICHTING KATHOLIEKE UNIVERSITEIT BRABANT
UTR	TARTU ULIKOOL
XLB	XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO
ATC	ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS
ENG	ENGINEERING - INGEGNERIA INFORMATICA SPA
UST	UNIVERSITAET STUTTGART
PRQ	PRAQMA A/S

The RADON project (January 2019 - June 2021) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825040

Executive summary

This deliverable presents the RADON Constraint Definition Language (CDL) and its associated Verification Tool (VT). The CDL can be used to encode both functional and non-functional requirements. The VT is able to check whether a given RADON model conforms to a given CDL specification and can suggest corrections to RADON models to rectify any detected inconsistencies.

Glossary

ASP	Answer Set Programming
AWS	Amazon Web Services
CDL	Constraint Definition Language
FaaS	Function as a Service
TOSCA	Topology and Orchestration Specification for Cloud Applications
VT	Verification Tool

Table of contents

1. Introduction

The RADON Constraint Definition Language (CDL) provides a way of expressing the high level functional and non-functional requirements of a FaaS architecture. The associated Verification Tool (VT) will have three main modes of execution: (1) Verification; (2) Correction; and (3) Learning; these three modes are detailed below.

1) **Verification**. In this mode, the tool checks whether a given RADON model complies with the constraints expressed in a given CDL specification.
2) **Correction**. In this mode, the tool searches for a modification of a (non-compliant) RADON model, in order to make it comply with the constraints in a given CDL specification.
3) **Learning**. In this mode, the tool is given examples of valid and invalid RADON models, or execution traces, and a partial CDL specification. The tool then searches for an extension of the CDL specification which rules out all of the invalid examples, which maintaining consistency with the valid examples.

This deliverable presents a first version of the CDL and a command line version of the VT which supports the first two modes of execution.

1.1 Deliverable Objectives

This deliverable has three main objectives:

1. Provide a formal definition of the language and semantics of the Constraint Definition Language.
2. Introduce a method for automatically verifying that a given RADON model complies with a given CDL specification.
3. Introduce a method for searching for a correction of a non-compliant RADON model, given a CDL specification.

1.2 Overview of Main Achievements

In the first year of the project, the main achievements have been to develop the CDL, which is capable of expressing requirements, and to develop a command line version of the VT. This command line VT supports the first two modes of execution (verification and correction), and can detect inconsistencies with the requirements expressed in the CDL. We have developed a new notion of "inconsistency specification", which allows a RADON user to encode potential issues such as conditions representing RADON models which may be at risk of race-conditions, deadlocks and execution loops, or RADON models which may encounter run-time errors. In Section 5, we demonstrate how each of the types of requirements identified in Deliverable 2.1 can be expressed as a general inconsistency specification. These general inconsistency specifications are available for RADON users to import from a CDL "standard library", enabling non-advanced

users to easily express their requirements. The advantage of this general approach (over hard-coding concepts such as race-conditions in the VT) is that advanced users can define their own inconsistencies, thus allowing the functionality of the final tool to be extended.

1.3 Structure of the Document

The remainder of the document is structured as follows. Section 2 recalls the requirements which apply specifically to the CDL and VT. Section 3 formalises the CDL and gives a full example of how to encode a scenario based on the RADON toy example as a CDL specification. Section 4 describes how to represent various verification tasks in the CDL, such as searching for race-conditions, deadlocks and execution loops. Section 5 presents the VT and explains the methods used for its first two modes of execution (verification and correction). Finally, Section 6 concludes the deliverable and discusses the remaining work.

2. Requirements

The RADON requirements analysis was presented in Deliverable 2.1. The subset of these requirements that have been used to guide the development of the Constraint Definition Language and Verification Tool is summarised in Table 1.

Table 1. RADON requirements which are relevant to the CDL and VT.

ID	Category	Title	Description
R-T4.1-1	Must have	CDL: Pre/post conditions	The CDL must be able to express pre/post conditions of serverless functions regarding security/performance.
R-T4.1-2	Must have	CDL: Hard constraints	The CDL must be able to express hard constraints on the required security/performance.
R-T4.1-3	Must have	CDL: Architectural pattern constraints	The CDL must be able to express hard constraints on the architectural patterns of sets of nodes.
R-T4.1-4	Should have	CDL: Soft constraints	The CDL should be able to express soft constraints on the required security/performance.
R-T4.1-5	Must have	VT: Hard constraints	The verification tool must be able to check that hard constraints are guaranteed to be satisfied by a given RADON model.
R-T4.1-6	Must have	VT: Hard constraints (real time)	This computation to check whether hard constraints are satisfied by a RADON model should return within a predefined maximum time.
R-T4.1-7	Should have	VT: Race conditions, loops and deadlocks	The verification tool should be able to check for the existence of potential race conditions and/or execution loops and related deadlocks that could happen.
R-T4.1-8	Should have	VT: Race conditions, loops and deadlocks (real time)	The computation to check for race conditions, execution loops and deadlocks should return within a predefined maximum time.
R-T4.1-9	Could have	VT: Correction	Given a RADON model that violates some constraints, the tool could provide corrections to the

			RADON model to ensure that it complies with the constraints.
R-T4.1-10	Could have	VT: Correction (real time)	This computation should complete within a predefined maximum time.
R-T4.1-11	Could have	VT: Verification of space of models	Given a space of possible RADON models (in a dynamic situation, new devices such as robotic assistants may be added/taken away), the tool could verify that any RADON model in the space complies with a set of hard constraints.
R-T3.2-12	Could have	VT: Soft constraints, search	Given a space of possible RADON models, the tool could compute an optimal RADON model with respect to CDL constraints. The computation for this may take place offline.
R-T3.2-13	Could have	VT: Soft constraints, improvement	Given a compliant sub-optimal RADON model, the tool could provide suggestions, which would improve its score with respect to the CDL soft constraints, while keeping the change to the original RADON model as small as possible. This computation should be fast enough to be used by a user interactively.

This deliverable at least partially addresses every requirement in the table, other than those on soft constraints (R-T4.1-4, R-T3.2-12 and R-T3.2-13). We refer back to this table through the rest of this deliverable, highlighting when a particular requirement is addressed. The remaining requirements will be addressed in a future deliverable.

3. Constraint Definition Language

This section defines the syntax and semantics of the Constraint Definition Language (CDL) and gives an example CDL specification.

3.1 Primitives

Firstly, we define the primitives, **word**, **variable**, **number** and **string** which are used in the rest of the grammar.

```
word -> [a-zA-Z][a-zA-Z0-9_\-]*
variable -> $[a-zA-Z][a-zA-Z0-9_]*
string -> '"' [^"]* '"'
number -> sign integer decimal

sign ->
sign -> '-'

integer -> [0-9]
integer -> [1-9] [0-9]+

decimal ->
decimal -> '.' [0-9]+
```

3.2 Entities and Values

A value takes many forms:

```
value -> number | variable | string |
         entity | set | array | condition | arithmetic_expression
```

The concepts of numbers, variables and strings were defined in the previous section. In the remainder of this section, we define entities, sets, arrays and conditions.

3.2.1 Entities

An entity is a dot separated list of words, e.g. **bucket** or **bucket.region**, or a variable followed by a dot separated list of words, e.g. **$b** or **$b.region**.

```
entity -> word | variable
entity -> entity '.' word
```

3.2.2 Sets

The CDL supports explicitly defined sets.

```
set -> '{' '}'
set -> '{' list_of_elements '}'
list_of_elements -> value
list_of_elements -> value ',' list_of_elements
```

In addition to explicit sets, entities may also represent sets.

```
set -> entity
```

3.2.2.1 Set Operations

Set operations can also be used to manipulate existing sets.

```
set -> set '.' 'such_that' '(' condition ')'
set -> set '.' 'union' '(' set ')'
set -> set '.' 'intersection' '(' set ')'
set -> set '.' 'subtract' '(' set ')'
set -> 'closure' '(' set ',' variable ',' variable ':' set ','
       condition ')'
```

The semantics of these set operations is defined as standard. So for any sets S_1 and S_2 and any condition `C`:

- S_1`.such_that(C)` returns those elements of S_1 that satisfy `C`.
- S_1`.union(`S_2`)` returns the union of S_1 and S_2.
- S_1`.intersection(`S_2`)` returns the intersection of S_1 and S_2.
- S_1`.subtract(`S_2`)` returns the elements of S_1 that are not in S_2.
- For any two variables V_1 and V_2, `closure(`S_1`,`V_1`,`V_2`:`S_2`,C)` returns the set constructed by starting with S_1 and iteratively adding the elements V_2 from S_2 that satisfy the condition `C`, where V_1 represents the initial set in each iteration.

Just as with mathematical sets, the order of elements within a set does not matter, and duplicate elements have no effect (so `{a, b} = {b, b, a}`). Finally, we can refer to the size of a set.

```
value -> set '.' 'size'
```

This expression evaluates to the number of distinct elements in the set.

3.2.3 Arrays

The CDL supports explicitly defined arrays.

```
array -> '[' ']'
array -> '[' list_of_elements ']'
```

Unlike sets, the order of elements, and duplicate elements do matter for arrays (so `[a, a, b]` ≠ `[a, b]` ≠ `[b, a]`).

3.2.3.1 Array Operations

Array operations can also be used to access and constrain arrays.

```
value -> array '[' value ']'
```

If the value evaluates to an integer `i`, this expression evaluates to the i^{th} element of the array.

```
value -> array '.' 'size'
```

This expression evaluates to the number of elements in the array.

3.2.4 Conditions

The most basic condition supported by the CDL is a binary comparison.

```
condition -> value comparison value
comparison -> '=' | '!=' | '<' | '<=' | '>' | '>='
```

For any two numeric values, these comparisons are defined as standard (so `1 < 2` returns true, but `2 < 1` returns false). If either of the two values is not numeric, the comparison returns false (unless the operation is `'='` or `'!='`).

Conditions can also be expressed on sets.

```
condition -> set '.' 'includes' '(' value ')'
```

`S.includes(E)` evaluates to true iff `S` is a set and `E` is in `S`.

```
condition -> set '.' 'empty' '(' ')'
```

`S.empty()` evaluates to true iff `S` is a set `S` is empty.

```
condition -> set '.' 'subset' '(' set ')'
```

S_1`.subset(`S_2`)` evaluates to true iff S_1 and S_2 are both sets and S_1 is a subset of S_2.

3.2.4.1 Complex Conditions

```
condition -> 'NOT' condition
condition -> condition 'AND' condition
condition -> condition 'OR'  condition
condition -> condition '=>'  condition
condition -> quantifier '(' variable ':' entity ',' condition ')'

quantifier -> 'FORALL' | 'EXISTS'
```

3.2.5 Arithmetic

The CDL also supports the following arithmetic expressions:

```
condition -> '(' value binop value ')'
binop -> '+' | '-' | '*' | '/'
```

Note that the parentheses may be omitted, in which case, the standard operator precedences apply.

3.3 CDL Specifications

A CDL specification consists of a set of statements and comments.

```
comment -> '#'[^\n]*
cdl_specification ->
cdl_specification -> comment cdl
cdl_specification -> statement ';' cdl_specification
```

The rest of the section defines CDL statements. In addition to statements for importing external files, there are two basic forms of statement. The first asserts that an entity must be equal to a value and the second asserts that a condition must hold. These assertions are called *hard constraints*.

```
statement -> entity '=' value
statement -> 'ASSERT' '(' condition ')'
```

In addition to conforming to the grammar so far, CDL statements must not contain any variable which is not bound to a quantifier (i.e. no variable can occur outside of a **FORALL** or **EXISTS** where it is the variable before the '**:**'). For example, **ASSERT(FORALL($X : s, $X > 0 AND $X < 20))** is a valid CDL statement, but **ASSERT($X > 0 AND FORALL($X : s, $X < 20))** is invalid, as the first **$X** is not bound to a quantifier.

3.3.1 Semantics

A CDL-interpretation $\mathbf{I}$ consists of a set of assignments $\mathbf{I}^{\text{assignments}}$ of the form $\mathbf{v_1{=}v_2}$ and a set $\mathbf{I}^{\text{constraints}}$ of CDL conditions. A CDL-interpretation $\mathbf{I_2}$ is said to be an extension of $\mathbf{I}$ (written $\mathbf{I} \subseteq \mathbf{I_2}$) if and only if $\mathbf{I}^{\text{assignments}} \subseteq \mathbf{I_2}^{\text{assignments}}$ and $\mathbf{I}^{\text{constraints}} \subseteq \mathbf{I_2}^{\text{constraints}}$.

A CDL-interpretation $\mathbf{I}$ is said to be complete iff each of the following properties holds:

1. For each $\mathbf{v}$ in the language, $\mathbf{v{=}v} \in \mathbf{I}^{\text{assignments}}$.
2. For each expression $\mathbf{e}$ in the language such that $\mathbf{e}$ evaluates to $\mathbf{v}$, $\mathbf{e{=}v} \in \mathbf{I}^{\text{assignments}}$. For example $\mathbf{\{a\}.union(\{b\})=\{a,b\}} \in \mathbf{I}^{\text{assignments}}$.
3. For each $\mathbf{v_1{=}v_2} \in \mathbf{I}^{\text{assignments}}$, $\mathbf{v_2{=}v_1} \in \mathbf{I}^{\text{assignments}}$.
4. For each pair $\mathbf{v_1{=}v_2}$ and $\mathbf{v_2{=}v_3} \in \mathbf{I}^{\text{assignments}}$, $\mathbf{v_1{=}v_3} \in \mathbf{I}^{\text{assignments}}$.
5. For each pair $\mathbf{v_1{=}v_2}$ and $\mathbf{v_2.property{=}v_3} \in \mathbf{I}^{\text{assignments}}$, $\mathbf{v_1.property{=}v_3} \in \mathbf{I}^{\text{assignments}}$.

6. For each $v_1=v_2 \in I^{conditions}$, $v_1=v_2 \in I^{assignments}$.
7. For each $NOT\ NOT\ C \in I^{conditions}$, $C \in I^{conditions}$.
8. For each $C_1\ AND\ C_2 \in I^{conditions}$, $C_1 \in I^{conditions}$ and $C_2 \in I^{conditions}$.
9. For each $C_1\ OR\ C_2 \in I^{conditions}$, either $C_1 \in I^{conditions}$ or $C_2 \in I^{conditions}$.
10. For each $C_1\ =>\ C_2 \in I^{conditions}$, $NOT\ C_1\ OR\ C_2 \in I^{conditions}$.
11. For each $NOT\ (C_1\ AND\ C_2) \in I^{conditions}$, $NOT\ C_1\ OR\ NOT\ C_2 \in I^{conditions}$.
12. For each $NOT\ (C_1\ OR\ C_2) \in I^{conditions}$, $NOT\ C_1\ AND\ NOT\ C_2 \in I^{conditions}$.
13. For each $NOT\ (C_1\ =>\ C_2) \in I^{conditions}$, $C_1\ AND\ NOT\ C_2 \in I^{conditions}$.
14. For each $NOT\ v_1\ =\ v_2 \in I^{conditions}$, $v_1\ !=\ v_2 \in I^{conditions}$.
15. For each $FORALL(V:S,\ C) \in I^{conditions}$, each $S=\{s_1,...,s_n\} \in I^{assignments}$: for each $i \in [1,n]$, $C[V\backslash s_i] \in I^{conditions}$, where $C[V\backslash s_i]$ is constructed by replacing all occurrences of V in C with s_i.
16. For each $EXISTS(V:S,\ C) \in I^{conditions}$, there is at least one $S=\{s_1,...,s_n\} \in I^{assignments}$ and at least one $i \in [1,n]$ such that $C[V\backslash s_i] \in I^{conditions}$.
17. For each $NOT\ FORALL(V:S,\ C) \in I^{conditions}$, $EXISTS(V:S,\ NOT\ C) \in I^{conditions}$.
18. For each $NOT\ EXISTS(V:S,\ C) \in I^{conditions}$, $FORALL(V:S,\ NOT\ C) \in I^{conditions}$.
19. For each $S.empty() \in I^{conditions}$, $S=\{\} \in I^{conditions}$.
20. For each $S.includes(E) \in I^{conditions}$, $EXISTS(V:S,\ V=E) \in I^{conditions}$.
21. For each $S_1.subset(S_2) \in I^{conditions}$, $FORALL(V:S_2,\ S_1.includes(V)) \in I^{conditions}$.
22. For any comparison condition C in $I^{conditions}$, the arguments of C are assigned to at least one pair of numbers respecting the comparison.
23. For any arithmetic expression E occurring anywhere in I, the arguments of E are assigned to at least one pair of numbers.

A CDL-interpretation I is said to be consistent iff there is at least one complete extension I_c of I such that each of the following properties holds:

1. There is no pair of assignments $v_1=v_2$ and $v_1=v_3 \in I^{assignments}$ such that v_2 and v_3 are both value types (numbers, strings, sets or arrays) such that $v_2 != v_3$.
2. There is no assignment $v_1=v_2 \in I^{assignments}$ such that $v_1!=v_2 \in I^{conditions}$.
3. There is no assignment $v_1=v_2 \in I^{assignments}$ where v_1 and v_2 are words and there is another assignment $v_1.prop=v_3 \in I^{assignments}$.
4. There is no assignment $v.prop1=v.prop2 \in I^{assignments}$ where $prop1$ and $prop2$ are words.

A CDL-interpretation I is said to satisfy a set of CDL statements S if and only if each of the following statements holds:

1. For each equality statement $v_1=v_2 \in S$, $v_1=v_2 \in I^{assignments}$.
2. For each assertion statement $ASSERT(C) \in S$, $C \in I^{conditions}$.

A CDL-interpretation is said to be a CDL-model of a set of CDL statements **S** if and only if it is complete and consistent and it satisfies **S**.

3.4 Importing RADON models

Another basic CDL statements is to import another file.

```
statement -> 'import' '"' file_path '"'
```

There are two accepted file extensions. If the file has the extension ".cdl" it is imported and interpreted as part of the CDL specification. If the file has the extension ".yml", it is first translated into a set of CDL statements, which are then treated as part of the specification. For example, consider the following partial YAML RADON model (where **[...]** denotes an omission from the full model):

```
tosca_definitions_version: tosca_simple_yaml_1_2
description: A toy example to get started with RADON.
imports:
  - file: radon_artifact_types.yml
  - file: radon_relationship_types.yml
  - file: radon_node_types.yml
  - file: radon_policy_types.yml
  - file: radon_private_types.yml

topology_template:
  [...]
  node_templates:
    uploads_1:
      type: radon.nodes.ObjectStorage
      properties:
        name: Uploads
      requirements:
        - host:
            node: amazon_aws_1
            relationship: uploads_host
        - invocation:
            node: index_photo
            relationship: uploads_invocation
      interfaces:
        Standard:
          inputs:
            host_access_key: { get_property: [ HOST, access_key ] }
            host_secret_access_key: { get_property: [ HOST, secret_access_key
] }
            host_region: { get_property: [ HOST, region ] }

    uploads_2:
      type: radon.nodes.ObjectStorage
      properties:
```

```
        name: Uploads
      requirements:
        - host:
            node: amazon_aws_2
            relationship: uploads_host
        - invocation:
            node: index_photo
            relationship: uploads_invocation
      interfaces:
        Standard:
          inputs:
            host_access_key: { get_property: [ HOST, access_key ] }
            host_secret_access_key: { get_property: [ HOST, secret_access_key
] }
            host_region: { get_property: [ HOST, region ] }

   amazon_aws_1:
      type: radon.nodes.AmazonAWS
      properties:
        access_key: { get_input: aws_access_key }
        secret_access_key: { get_input: aws_secret_access_key }
        region: eu-west-1

    amazon_aws_2:
      type: radon.nodes.AmazonAWS
      properties:
        access_key: { get_input: aws_access_key }
        secret_access_key: { get_input: aws_secret_access_key }
        region: eu-west-2
```

The RADON model above is translated into the following set of CDL statements:

```
import "radon_artifact_types.yml";
import "radon_relationship_types.yml";
import "radon_node_types.yml";
import "radon_policy_types.yml";
import "radon_private_types.yml";

uploads_1.type = radon.nodes.ObjectStorage;
uploads_1.name = Uploads;
uploads_1.host.node = amazon_aws_1;
uploads_1.host.relationship = uploads_host;
uploads_1.invocation.node = index_photo;
uploads_1.invocation.relationship = uploads_invocation;
uploads_2.type = radon.nodes.ObjectStorage;
uploads_2.name = Uploads;
uploads_2.host.node = amazon_aws_2;
uploads_2.host.relationship = uploads_host;
```

```
uploads_2.invocation.node = index_photo;
uploads_2.invocation.relationship = uploads_invocation;
amazon_aws_1.type = radon.nodes.AmazonAWS;
amazon_aws_1.region = eu-west-1;
amazon_aws_2.type = radon.nodes.AmazonAWS;
amazon_aws_2.region = eu-west-2;
```

Note that not all parts of the RADON model are currently translated. For each **`node_template`**, the **`type`**, **`properties`**, and **`requirements`** are translated, while the interfaces (for example) are ignored. We will expand the subset of the RADON model that is translated as required for the use cases.

3.5 Syntactic Sugar

In addition to the CDL introduced earlier in this section, there are several extra language constructs to compact certain expressions in the CDL.

3.5.1 Array Declarations

Rather than explicitly defining an array, it is also possible to define it based on its name and size alone, without needing to specify the individual elements.

```
statement -> entity '[' value ']'
```

If the value **`v`** evaluates to an integer **`n`**, then the statement **`t[v]`** is equivalent to the statement **`t = [t[1], ..., t[n]]`**.

3.5.2 Set Declarations

Often it is useful to define ranges of integers, without explicitly defining the set of integers in the range (for instance, **`{1..5}`** rather than **`{1, 2, 3, 4, 5}`**). This is particularly useful when one of the bounds of the range is a variable.

```
set -> '{' value '..' value '}'
```

If the values v_1 and v_2 evaluates to the integers n_1 and n_2, then this is equivalent to the set $\{n_1, (n_1 + 1) ..., (n_2 - 1), n_2\}$.

3.5.3 Picking Elements from Sets

Often when writing CDL specifications, it is useful to draw an arbitrary element **`e`** from a set **`S`** and perform some reasoning. In the CDL this can be achieved with the assertion **`ASSERT(S.includes(e))`**, but this might seem an unnatural expression, as this may be the first occurence of **`e`**, and developers may find it easier to think of an element of **`S`** being "assigned" to **`e`**. The CDL allows such an "assignment" operation:

```
statement -> entity '<-' set
```

In reality, the statement `e <- S` is equivalent to the assertion `ASSERT(S.includes(e))`, as in the CDL, there is no need to formally declare and assign a value to `e`, but the `<-` operator more naturally depicts the developer's intention.

3.6 Encoding the Toy Example as a CDL specification

We now encode an extension of the RADON toy example, in which there is a lambda function "create_thumbnails" and a set of S3 buckets. In this section, we extend the scenario with a simple constraint, stating that each country us only willing to store their own data in a restricted set of countries.

First, we import a TOSCA model:

```
import "outputs/service_template_instance.yml";
```

The file `"outputs/service_template_instance.yml"` is the RADON model given in Section 3.4. It is translated to CDL statements as shown in Section 3.4.

Next we define several concepts that are not in the TOSCA model, such as the geographical locations of regions, and the set of countries with whom each supported country is willing to share.

```
thumbnail_buckets = {uploads_1, uploads_2};

eu-west-1.hosted_in = ireland;
eu-west-2.hosted_in = uk;

supported_countries = { uk, us, canada, china, india };
uk.willing = { uk, us, canada, china, india, ireland};
us.willing = { us };
canada.willing = { uk, us, canada };
china.willing = { china };
india.willing = { india, uk };
```

We can also define properties of the create_thumbnails lambda function, such as its inputs and pre/postconditions. Note that where a condition should be satisfied both before and after the execution of a function, rather than declaring that it is both a pre and a postcondition, it is declared as an invariant condition.

```
create_thumbnails.inputs = {
  input.country_of_origin,
  input.thumbnail
};

create_thumbnails.pre_conditions = {
  supported_countries.includes(input.country_of_origin)
};

create_thumbnails.post_conditions = {
  EXISTS($B : thumbnail_buckets, $B.storage.includes(input.thumbnail))
```

```
};

create_thumbnails.invariant_conditions = {
  FORALL($B : thumbnail_buckets,
    $B.storage.includes(input.thumbnail)
    =>
    input.country_of_origin.willing.includes(
      $B.host.node.region.hosted_in
    )
  )
};
```

It is clear from the pre/postconditions that for each supported country, there must be at least one bucket contained in a country with whom the original country is willing to share, for if this is not the case then there will be valid inputs to create_thumbnails which meet the preconditions but for which the postconditions cannot be met. The next section shows a method for finding these cases automatically, using the verification tool, but they can also be ruled out explicitly, using assertions.

```
ASSERT(FORALL($C : supported_countries,
    EXISTS($B : thumbnail_buckets,
        $C.willing.includes($B.host.node.region.hosted_in)
    )
  )
};
```

This assertion statement rules out any TOSCA model that for which there is a supported country that is not willing to store their data in at least one bucket.

4 Defining a Verification Task

The verification tool can be used to search for many kinds of inconsistency in a model RADON model. The notion of an inconsistency is intended to be very flexible, meaning that users can define their own inconsistencies and use the verification tool to search for them. In this section, we introduce the language for defining inconsistencies, and demonstrate how this enables the tool to automatically identify potential race conditions, deadlocks, execution loops, and also express architectural requirements and detect potential runtime errors by testing for inconsistent pre/postconditions. Advanced RADON users may use this language to define their own notions of inconsistency, but to aid the less advanced users, the inconsistencies defined in this section will all be available to import from a built-in library in the CDL tool.

4.1 The Language of Inconsistency Specifications

Inconsistencies are defined as a block containing both normal CDL statements and extra statements reserved only for the definition of inconsistencies.

```
cdl_specification -> inconsistency ';' cdl_specification

inconsistency -> 'INCONSISTENCY' '{' inner_incon '}'

inner_incon ->
inner_incon -> cdl_statement ';' inner_incon
inner_incon -> 'timeline' entity '=' array ';' inner_incon
inner_incon -> 'timeline' entity '[' value ']' ';' inner_incon
```

A timeline is an (ordered) list of timepoints, representing runtime timepoints.

There is also an additional CDL-statement that can be used to declare which entities may change value over the course of a timeline (all other entity values are assumed to be fixed).

```
statement -> 'runtime_variable' entity
```

Timepoints can have various properties imposed on them using assertions. To achieve this, there are special conditions (denoted **`condition_extra`**) below which can only be used in assertions in inconsistency specifications.

```
condition_extra -> condition
condition_extra -> entity '.' satisfies '(' condition_extra ')'
condition_extra -> entity '.' consistent
condition_extra -> entity '.' inconsistent
inner_incon -> 'ASSERT' '(' condition_extra ')'
```

Note that if the entity in a **condition_extra** expression does not evaluate to a defined timepoint, then the condition evaluates to false.

Most inconsistency instances will define a series of timepoints which must be consistent, followed by a timepoint which is inconsistent. The idea is that the consistent timepoints depict a sequence of events which comply with the constraints (i.e. the behaviour is consistent with the requirements), but they lead to a timepoint which cannot possibly comply with the requirements.

Given a CDL-specification **S**, containing an inconsistency **INC**, the inconsistency instances of **INC** are of the form:

```
{
    I,
    TL_1 : [I_1,1, ..., I_1,n1],
    ...,
    TL_m : [I_m,1, ..., I_m,nm]
}
```

Where $TL_1, ..., TL_m$ are the timelines in **INC** and the following properties all hold:

1. I is a CDL-model of $S \cup INC_{inner}$ (where INC_{inner} is the set of CDL statements in **INC**).
2. For each $i \in [1,m]$, $n_i = |TL_i|$.
3. For each $i \in [1,m]$, $j \in [1,n_i]$, $I \subseteq I_{i,j}$ and for each condition of the form $TL_i[j].satisfies(C) \in I^{constraints}$, $C \in I_{i,j}{}^{constraints}$.
4. For each $i \in [1,m]$, $j \in [1,n_i]$, $I_{i,j}$ is consistent iff $TL_i[j].consistent \in I^{constraints}$.
5. For each $i \in [1,m]$, $j \in [1,n_i]$ such that $TL_i[j].consistent \in I^{constraints}$, $I_{i,j}$ is complete.
6. For each entity **e** that is not a runtime variable, the value of **e** is the same for every timepoint within the same timeline.

The set of inconsistency instances of **S** is the set containing all inconsistency instances of all inconsistencies in **S**.

4.2 Example Inconsistency Specifications

In this section, we give examples of inconsistency specifications. The inconsistency specifications in sections 4.2.1—4.2.4 are available for users to import in CDL specifications as part of a CDL standard library[1].

[1] Note that imports from the standard library are of the form **import <...>** rather than **import "..."**).

4.2.1 Verification of Preconditions and Postconditions

The following inconsistency definition searches for potential runtime errors, by checking whether there is any potential scenario where a function's preconditions are all satisfied, the function is then called, and it is impossible for the function to satisfy its postconditions. An instance of this inconsistency occurs when there is a timeline of two points such that the first timepoint satisfies a function's preconditions (and invariant conditions) but in order for the second timepoint to satisfy the function's postconditions (and invariant conditions) it must be inconsistent.

```
# This inconsistency definition is available to import from the CDL
# standard library as <pre_post_conditions.cdl>.

INCONSISTENCY pre_post_conditions {
    timeline t = [t1, t2];
    f <- functions;

    ASSERT(t1.satisfies(FORALL($PC : f.pre_conditions, $PC)));
    ASSERT(t1.satisfies(FORALL($PC : f.invariant_conditions, $PC)));

    ASSERT(t2.satisfies(FORALL($PC : f.post_conditions, $PC)));
    ASSERT(t2.satisfies(FORALL($PC : f.invariant_conditions, $PC)));

    ASSERT(t1.consistent);
    ASSERT(NOT t2.consistent);
};
```

4.2.1.1 Extension of the toy example

In Section 3.6, the toy example specification required that there was at least one bucket in which each of a set of countries was willing to store their data. This was a fairly obvious consequence of the pre/postconditions of the create_thumbnails lambda functions, and so we hard-coded it as an assertion in the CDL-specification. In other situations, such consequences may not be immediately obvious to a RADON user. We can use the **`pre_post_conditions`** inconsistency definition to detect such cases. In fact, if we add the following runtime_variable declarations to the toy example specification in Section 3.6 and also import the **`pre_post_conditions`** inconsistency definition, the verification tool can find exactly the same inconsistencies as it does with the hard-coded assertion.

```
runtime_variable uploads_1.storage;
runtime_variable uploads_2.storage;
```

When the verification tool is run on this task, it finds two instances of this inconsistency, corresponding to the two supported countries which would be unable to store their data (the US and China).

4.2.2 Searching for Deadlocks

Deadlocks occur when a set of threads T are all waiting to use a resource that is currently locked by a different thread in T. To reason about the behaviour of functions, we need some basic information about which resources they require and in what order. Consider a scenario where we have two functions, **f1** and **f2**, which both require the resources **a** and **b**, but in different orders.

```
# This file is available in the Verification Tool repository, as
# "examples/deadlocks.cdl".

import <deadlocks.cdl>;

all_functions = {fn1, fn2};
resources = {a, b};
fn1.steps[4];
fn2.steps[4];

fn1.steps[1].required_locks = {a};
fn1.steps[2].required_locks = {b};
fn1.steps[3].required_locks = {};
fn1.steps[4].required_locks = {};

fn1.steps[1].released_locks = {};
fn1.steps[2].released_locks = {};
fn1.steps[3].released_locks = {b};
fn1.steps[4].released_locks = {a};

fn2.steps[1].required_locks = {b};
fn2.steps[2].required_locks = {a};
fn2.steps[3].required_locks = {};
fn2.steps[4].required_locks = {};

fn2.steps[1].released_locks = {};
fn2.steps[2].released_locks = {};
fn2.steps[3].released_locks = {a};
fn2.steps[4].released_locks = {b};
```

If two threads call **fn1** and **fn2** concurrently, the following sequence may occur

```
–> THREAD 1 calls fn1 and THREAD 2 calls fn2.

THREAD 1                                    THREAD 2

–> locked_resources = {}                    –> locked_resources = {}

Execute fn1.step[1]
–> locked_resources = {a}
                                            Execute fn2.step[2]
                                            –> locked_resources = {b}

Execute fn1.step[2]                         Execute fn2.step[2]
–> cannot execute until THREAD 2            –> cannot execute until THREAD 2
releases b.                                 releases a.

–> Neither thread can make progress, as both threads are waiting for a
resource that is locked by the other. This situation is called a deadlock.
```

The inconsistency definition below can be used to find deadlocks caused by two threads.

```
# This inconsistency definition is available to import from the CDL
# standard library as <deadlocks.cdl>.

INCONSISTENCY deadlock {

  f1 <- all_functions;
  f2 <- all_functions;
  functions = {f1, f2};

  # This definition takes a "guess and check approach". It guesses which
  # step of each function is blocked, which resource it is waiting for,
  # which function it is locked by, and at which step it was locked. It then
  # checks that this guess corresponds to a deadlock.
  ASSERT(FORALL($F : functions,

    # Guess which step is blocked for $F.
    blocked_step[$F] <- {1..$F.steps.size}
    AND

    # Guess which resource $F is waiting for.
    blocked_resource[$F] <- $F.steps[blocked_step[$F]].required_locks
    AND

    # Guess which function locked the resource $F is waiting for.
    blocked_by[$F] <- functions
    AND

    # Guess the step the resource $F is waiting for was locked.
    lock_time[blocked_by[$F]] <- {1..blocked_by[$F].steps.size}
```

```
    AND

    # Check that the locked resource was required by the other function at
    # the correct time.
    blocked_by[$F].steps[lock_time[blocked_by[$F]]]
      .required_locks.includes(blocked_resource[$F])
    AND

    # Check that the resource was locked before $F is blocked.
    (lock_time[$F] < blocked_step[$F])
    AND

    # Check that the resource was not released before the blocked step.
    FORALL($i : {1..blocked_by[$F].steps.size},
      (($i < blocked_step[blocked_by[$F]]) AND ($i > lock_time[$F]))
      =>
      (NOT blocked_by[$F].steps[lock_time[blocked_by[$F]]]
             .released_locks.includes(blocked_resource[$F])
      )
    )
  ));
};
```

This inconsistency could be generalised to a set of threads of size **n**, by replacing **f1** and **f2** with an array **f** of size **n**. If this inconsistency is imported into the **"examples/deadlocks.cdl"** file, then the VT can find the deadlock discussed at the beginning of this subsection.

4.2.3 Searching for Race Conditions

Race conditions occur when two threads access the same resources concurrently, and one thread (T_1) overwrites a resource that the other thread (T_2) has previously read from, before T_2 has completed. For example, a function **f** may be used to increase a person's bank balance by £100. The function **f** may work by reading the bank balance, increasing it by 100 and then writing the new balance, as depicted below:

```
void f() {
    b = balance.read();
    b += 100;
    balance.write(b);
}
```

If two threads call **f** concurrently, the following sequence may occur (assuming that the initial balance was £2306.16).

```
–> The user's initial bank balance is £2306.16

THREAD 1                                    THREAD 2
b = balance.read();
–> b1 = 2306.16
                                            b = balance.read();
                                            –> b2 = 2306.16


b += 100;
–> b1 = 2406.16
                                            b += 100;
balance.write(b);                           –> b2 = 2406.16

                                            balance.write(b);

–> The user's final bank balance is £2406.16, even though f() has been called
twice.
```

Race conditions cause non-deterministic behaviour. We can therefore detect them by searching for two timelines where the same two threads have been executed, with different outcomes.

```
# This inconsistency definition is available to import from the CDL
# standard library as <race_conditions.cdl>.

INCONSISTENCY race_condition {
  timeline t1[1];
  timeline t2[1];

  runtime_variables = {thread1, thread2, shared_variable};

  max_time_point = thread1.io_times.size + thread2.io_times.size;

  ASSERT(FORALL($c : initial_conditions, $c));

  ASSERT(FORALL($i : {1..thread1.io_times.size},
    thread1.io_times[$i] <- {1..max_time_point}
  ));

  ASSERT(FORALL($i : {1..thread2.io_times.size},
    thread2.io_times[$i] <- {1..max_time_point}
  ));

  ASSERT(t1[1].consistent);
  ASSERT(t2[1].consistent);

  ASSERT(FORALL($x : range,
    t1[1].satisfies((shared_variable[max_time_point] != $x))
    OR
```

```
    t2[1].satisfies((shared_variable[max_time_point] != $x))
  ));

  ASSERT(FORALL($s : thread1.steps, $s));
  ASSERT(FORALL($s : thread2.steps, $s));

  ASSERT(FORALL($t : {2..max_time_point},
    EXISTS($f : {thread1, thread2}, $f.writes.includes($t))
    OR
    (shared_variable[$t] = shared_variable[$t-1])
  ));

};
```

We can use this inconsistency specification to detect the sequence presented at the start of this subsection, which demonstrated a race condition. To do so, the user must describe the behaviour of the threads. This can be done as follows.

```
import <race_conditions.cdl>;

# define the shared variable, and its range of values.
shared_variable = balance;
range = {1..5};

thread1.io_times[2];
thread2.io_times[2];

thread1.steps = {
  (thread1.b_1 = balance[thread1.io_times[1]]),
  (thread1.b_2 = thread1.b_1 + 1),
  (balance[thread1.io_times[2]] = thread1.b_2)
};
ASSERT((thread1.io_times[1] < thread1.io_times[2]));

thread2.steps = {
  (thread2.b_1 = balance[thread2.io_times[1]]),
  (thread2.b_2 = thread2.b_1 + 1),
  (balance[thread2.io_times[2]] = thread2.b_2)
};
ASSERT((thread2.io_times[1] < thread2.io_times[2]));

thread1.writes = {thread1.io_times[2]};
thread2.writes = {thread2.io_times[2]};

initial_conditions = { (balance[1]=1) };
```

In the above specification, the RADON user needs to declare that **`balance`** is a shared variable (and should be monitored for race conditions). The specification could be generalised to multiple

shared variables by instead using a set of shared variables. Each thread has an **io_times** array, which denote the times at which the thread interacts with a shared variable. The inconsistency specification essentially searches for two different assignments to the **io_times** resulting in different final values for the shared variable. For instance, **thread1.io_times=[1,2]** and **thread2.io_times=[3,4]** would result in **balance** being **3**, but **thread1.io_times=[1,4]** and **thread2.io_times=[2,3]** would result in **balance** being **2**. If the verification tool is run with this specification, it will find this race condition.

4.2.4 Searching for Execution Loops

An execution loop can occur when a function can call itself (either directly, or indirectly). This can be useful, but it can also be problematic, as unless careful consideration is given to ensure that a termination condition is guaranteed to eventually be met, they can lead to infinite computation loops, which never terminate. One way of avoiding infinite loops is to forbid execution loops altogether. This can be achieved with the following verification specification.

```
# This inconsistency definition is available to import from the CDL
# standard library as <execution_loops.cdl>.

INCONSISTENCY execution_loop {
  loop_length <- {2..functions.size};
  fn[1] <- functions;
  ASSERT(FORALL($i : {1..loop_length},
    fn[$i+1] <- fn[$i].calls
  ));
  fn[loop_length+1] = fn[1];
};
```

Another way to ensure termination is to guarantee define a variant condition for any control loop, which defines a monotonically increasing or decreasing quantity, with a finite set of possible values. We have not explored this approach yet, but may do so in future work.

4.2.5 Verifying a space of RADON models

So far, each of the inconsistencies we have defined has been with respect to some fixed RADON model. In fact, the verification tool can also be used to check whether any RADON model in a space of possible models has any of the above inconsistencies. This is achieved by defining the space of extensions to a partial model in the main CDL specification. For instance, if we take the toy example, and we want to reason about a setting where a particular country's sharing policy is extremely volatile, and may completely change at any time, rather than explicitly defining the countries with whom they are willing to share, we can define the constraints on the range of possibilities using assertion statements. In the example shown below, the UK is always willing to share with itself, but may also be willing to share with any of the other supported countries.

```
ASSERT(uk.willing_to_share.includes(uk));
```

5. Verification Tool

In this section, we discuss the two main execution modes of the Verification Tool that have already been implemented. These are: (1) `verification`; and (2) `correction`. Both modes make extensive use of Answer Set Programming (ASP) [Gelfond & Lifschitz, 1988; Brewka et al, 2011] to compute solutions. The reasons for choosing ASP were threefold:

1. ASP is a highly expressive formalism, which is capable of representing all the verification tasks discussed in Section 4.
2. There are highly efficient ASP solvers [Leone et al, 2006; Gebser et al, 2007], which the Verification Tool makes use of.
3. In future work, we will extend the Verification Tool to enable learning extra constraints, given a CDL specification. The recent development of systems for learning ASP [Ray 2009; Corapi et al, 2011; Law et al, 2014, 2015, 2018a] will enable learning these constraints. In particular, the ILASP (Inductive Learning of Answer Set Programs) is capable of learning any ASP program [Law et al, 2018b], resources permitting, and can therefore learn additional constraints in any of the verification scenarios discussed in Section 4.

In the rest of this section, we present the `verification` and `correction` modes of the verification tool.

5.1 Verifying a CDL Specification in ASP

The solutions of answer set programs are called *answer sets*. The intuition behind the Verification Tool is that we represent a CDL specification **S** as an ASP program $\mathbf{P_S}$ such that the inconsistency instances of **S** are captured by the answer sets of $\mathbf{P_S}$. We omit the full technical details of the ASP representation in this document, but if the reader is interested, they can produce the ASP representation using the command line Verification Tool with the flag `--asp-representation`. When run in without this flag, in `verification` mode, the Verification Tool searches for a single inconsistency instance of the input CDL specification.

5.1.1 Toy Example Verification Example

Consider the following CDL specification:

```
# Import the inconsistency specification defined in Section 4.2.1.
import <pre_post_conditions.cdl>

# Import the toy example RADON model given in Section 3.4.
# This RADON model has two buckets, one in the UK and one in Ireland.
import "toy_example_radon_model.yml"

thumbnail_buckets = {uploads_1, uploads_2};
```

```
eu-west-1.hosted_in = ireland;
eu-west-2.hosted_in = uk;

supported_countries = { uk, us, canada, china, india };
uk.willing = { uk, us, canada, china, india, ireland};
us.willing = { us };
canada.willing = { uk, us, canada };
china.willing = { china };
india.willing = { india, uk };

create_thumbnails.inputs = {
  input.country_of_origin,
  input.thumbnail
};

create_thumbnails.pre_conditions = {
  supported_countries.includes(input.country_of_origin)
};

create_thumbnails.post_conditions = {
  EXISTS($B : thumbnail_buckets, $B.storage.includes(input.thumbnail))
};

create_thumbnails.invariant_conditions = {
  FORALL($B : thumbnail_buckets,
    $B.storage.includes(input.thumbnail)
    =>
    input.country_of_origin.willing.includes(
      $B.host.node.region.hosted_in
    )
  )
};
```

If the Verification Tool is run in `verification` mode, it outputs the following:

```
============================== Inconsistency 1 ==============================

Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
  (t[1])  input.country_of_origin = us
  (t[1])  input.country_of_origin.willing = us.willing
  (t[1])  input.country_of_origin.willing = {us}
  (t[1])  input.country_of_origin is in supported_countries

============================== Inconsistency 2 ==============================
```

```
Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
  (t[1])  input.country_of_origin = china
  (t[1])  input.country_of_origin.willing = china.willing
  (t[1])  input.country_of_origin.willing = {china}
  (t[1])  input.country_of_origin is in supported_countries
```

The two inconsistency instances show the assumptions that need to be made to the consistent timepoints in the timeline (for this inconsistency specification **`t1`** must be consistent and **`t2`** must be inconsistent). In the first case, the assumptions are that **`f`** is equal to **`create_thumbnails`**, **`input.country_of_origin`** is equal to **`us`**, and neither thumbnail bucket contains **`input.thumbnail`**. This corresponds to the **`create_thumbnails`** function being called with a thumbnail which is not stored in either thumbnail bucket and originates from the U.S. The second inconsistency is similar, but where the thumbnail originates from China.

5.2 Correcting a RADON model

The `correction` mode of the Verification Tool searches within a user-defined space of possible corrections to the RADON model, which has no inconsistency instances.

5.2.1 Defining a Correction Space

Correction actions to a CDL specification come in three forms:

1. *Definitions*: these are new equality statements, which are added to the CDL specification.
2. *Insertions*: this is where a new element is added to a set.
3. *Deletions*: this is where an element is removed from a set.

In principle, the Verification Tool could search for a set of correction actions to a CDL specification within a completely unbounded search space. This is problematic for two main reasons: (1) a lack of efficiency; and (2) that some trivial solutions (such as deleting all a functions post-conditions) may be undesirable. For these reasons, the Verification Tool takes, as part of its input, a definition of a *correction space*, which defines the set of all permitted correction actions. These are of the form:

```
correction -> '@definable' entity ';'
correction -> '@extendable' entity ';'
correction -> '@revisable' entity ';'
```

Entities which are *definable* may occur on the left hand side of new equality statements. Note that if an entity **`e`** is definable, then **`e.`**$\mathtt{w}_1\mathtt{...w}_n$ (for any set of **`w`**'s) is also definable. Previous definitions of the form **`e.`**$\mathtt{w}_1\mathtt{...w}_n$ may also be overwritten.

Set entities which are *extendable* may have new elements added to them. To keep the correction search space finite, these elements must already occur somewhere in the task, either in the original specification, or as a new definable entity. Set entities which are *revisable* may have existing elements deleted from them.

5.2.2 The Correction Algorithm

The correction mode of the Verification Tool is an iterative process, formalised by Algorithm 1, which builds up a set of inconsistency instances, searching in each iteration for a correction of the CDL specification for which none of the inconsistency instances are inconsistency instances of the corrected CDL specification. It relies on two subprocedures, **`compute_correction`** and **`verify`**. The **`compute_correction`** procedure searches within a correction space for a correction of the CDL specification, such that none of a set of already computed inconsistency instances are inconsistency instances of the corrected CDL specification (if no such correction exists, the procedure returns **`nil`**). The **`verify`** procedure computes an inconsistency instance of a given CDL specification (if no such inconsistency instance exists, the procedure returns **`nil`**).

```
Algorithm 1:

correct(CDL_spec, correction_space) {

  corrected_spec = {}
  inconsistencies = {}

  while(true) {

    corrected_spec = compute_correction(
                       CDL_spec,
                       inconsistencies,
                       correction_space
                     );

    if(correction_result == nil) {
      return UNSATISFIABLE;
    }

    verification_result = verify(corrected_spec);
    if(verification_result) {
      return corrected_spec;
    }

    inconsistencies.insert(verification_result);

  }

}
```

Algorithm 1: The iterative correction algorithm.

Note that sometimes the **compute_correction** procedure may have several alternative corrections which correct the inconsistency instances. When this occurs **compute_correction** returns one of the corrections containing the fewest individual correction actions.

5.2.3 Toy Example Correction Example

Reconsider the toy example CDL specification in Section 5.1.1. The Verification Tool running in `verification` mode returns the two inconsistency instances, representing the U.S. and China inconsistency instances. In `correction` mode, the Verification Tool finds a correction which eliminates these two inconsistencies without creating any new inconsistency instances. In this section, we give examples of this using three different correction spaces.

5.2.3.1 Creating New Thumbnail Buckets

Consider the following correction space:

```
@extendable thumbnail_buckets;
@definable new_node1;
@definable new_node2;
```

When the Verification Tool is run with this correction space, the output is as follows:

```
========================== Inconsistency 1 ==========================

Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
  (t[1])  input.country_of_origin = us
  (t[1])  input.country_of_origin.willing = us.willing
  (t[1])  input.country_of_origin.willing = {us}
  (t[1])  input.country_of_origin is in supported_countries

============================ Correction =============================

new_node1.host.node.region.hosted_in = us;
thumbnail_buckets = { new_node1, uploads_1, uploads_2};

========================== Inconsistency 2 ==========================

Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
```

```
  (t[1])  input.country_of_origin = china
  (t[1])  input.country_of_origin.willing = china.willing
  (t[1])  input.country_of_origin.willing = {china}
  (t[1])  input.country_of_origin is in supported_countries

============================ Correction ============================

new_node1.host.node.region.hosted_in = us;
new_node2.host.node.region.hosted_in = china;
thumbnail_buckets = { new_node1, new_node2, uploads_1, uploads_2};

====================================================================

No further inconsistency found.
```

The output shows the iterative procedure followed by the `correction` algorithm. First, the U.S. inconsistency instance is found, and a correction is found from within the correction space. The first definable node **`new_node1`** is added to the set of **`thumbnail_buckets`**, and the location of its host is set to **`us`**, which repairs the inconsistency instance. This does not resolve the China inconsistency instance, and so this is found next, and a further correction is found. This now corresponds to two new thumbnail buckets, located in the U.S. and China.

5.2.3.2 Revising the Set of Supported Countries

Widening the space of possible corrections allows the Verification Tool to find simpler solutions. Consider the following correction space:

```
@extendable thumbnail_buckets;
@definable new_node1;
@definable new_node2;
@revisable supported_countries;
```

When the Verification Tool is run with this extended correction space, the output is as follows:

```
========================== Inconsistency 1 ==========================

Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
  (t[1])  input.country_of_origin = us
  (t[1])  input.country_of_origin.willing = us.willing
  (t[1])  input.country_of_origin.willing = {us}
  (t[1])  input.country_of_origin is in supported_countries

============================ Correction ============================

supported_countries = { canada, china, india, uk};
```

```
========================== Inconsistency 2 ==========================

Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
  (t[1])  input.country_of_origin = china
  (t[1])  input.country_of_origin.willing = china.willing
  (t[1])  input.country_of_origin.willing = {china}
  (t[1])  input.country_of_origin is in supported_countries

============================ Correction =============================

supported_countries = { canada, india, uk};

=====================================================================

No further inconsistency found.
```

The output shows that when the Verification Tool is run with this extended correction space, it finds simpler corrections. To correct the U.S. inconsistency instance, the U.S. is removed from the set of supported countries. This does not resolve the China inconsistency instance, and so this is found next, and a further correction is found. This now corresponds to both the U.S. and China being removed from the set of supported countries.

5.2.3.3 Revising the Set of Post-conditions

Consider the following correction space:

```
@extendable thumbnail_buckets;
@definable new_node1;
@definable new_node2;
@revisable supported_countries;
@revisable create_thumbnails.post_conditions;
```

When the Verification Tool is run with this correction space, the output is as follows:

```
========================== Inconsistency 1 ==========================

Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
  (t[1])  input.country_of_origin = us
  (t[1])  input.country_of_origin.willing = us.willing
  (t[1])  input.country_of_origin.willing = {us}
  (t[1])  input.country_of_origin is in supported_countries
```

```
============================== Correction ==============================

supported_countries = { canada, china, india, uk};

========================== Inconsistency 2 ==========================

Detected pre_post_conditions inconsistency. The following assertions are
sufficient to demonstrate the inconsistency:
  (base[0])  supported_countries = {uk, us, canada, china, india}
  (base[0])  uploads_1.storage = $B.storage ($B => uploads_1)
  (base[0])  uploads_2.storage = $B.storage ($B => uploads_2)
  (base[0])  f = create_thumbnails
  (t[1])  input.country_of_origin = china
  (t[1])  input.country_of_origin.willing = china.willing
  (t[1])  input.country_of_origin.willing = {china}
  (t[1])  input.country_of_origin is in supported_countries

============================== Correction ==============================

create_thumbnails.post_conditions = { };

========================================================================

No further inconsistency found.
```

The output shows that when the Verification Tool is run with this further extended correction space, it finds a much simpler solution, which is to delete the postcondition of the **`create_thumbnails`** function.

6. Conclusions

This deliverable has introduced the CDL, which can be used to express functional and non-functional requirements on a RADON model. We have also presented two modes of the VT, which enable RADON users to verify that a RADON model is consistent with a CDL specification, and to search for corrections of an inconsistent RADON model.

Table 2 shows an overview of the level of fulfillment for each of the agreed requirements. The labels specifying the "Level of compliance" are defined as follows:

(i) ✗ (unsupported): the requirement is not fulfilled by the current version

(ii) ✔ (partially-low supported): a few of the aspects of the requirement is fulfilled by the current version

(iii) ✔✔ (partially-high supported): most of the aspects of the requirement is fulfilled by the current version

(iv) ✔✔✔ (fully supported): the requirement is fulfilled by the current version). Afterwards, we discuss for each requirement briefly how it has been addressed by publishing this deliverable

Table 2. Achieved level of compliance to RADON requirements

Id	Requirement Title	Priority	Level of compliance
R-T4.1-1	CDL: Pre/post conditions	Must have	✔✔✔
R-T4.1-2	CDL: Hard constraints	Must have	✔✔
R-T4.1-3	CDL: Architectural pattern constraints	Must have	✔
R-T4.1-4	CDL: Soft constraints	Should have	✗
R-T4.1-5	VT: Hard constraints	Must have	✔✔
R-T4.1-6	VT: Hard constraints (real time)	Must have	✔✔
R-T4.1-7	VT: Race conditions, loops and deadlocks	Should have	✔✔✔
R-T4.1-8	VT: Race conditions, loops and deadlocks (real time)	Should have	✔✔
R-T4.1-9	VT: Correction	Could have	✔✔✔
R-T4.1-10	VT: Correction (real time)	Could have	✔✔
R-T4.1-11	VT: Verification of space of models	Could have	✔✔✔

R-T3.2-12	VT: Soft constraints, search	Could have	✗
R-T3.2-13	VT: Soft constraints, improvement	Could have	✗

We showed in Section 3.6 that pre and post conditions can be encoded in a CDL specification, showing full compliance with R-T4.1-1. We are mostly compliant with R-T4.1-2, in that the CDL can encode hard constraints (using the assertion statements defined in Section 3) on the required security/performance. One related aspect that the CDL does not yet represent is policy. For full compliance with R-T4.1-2, we intend to extend the CDL to represent policies. We have not yet demonstrated that we can represent architectural pattern constraints in the CDL, but the predicates we have defined should be sufficient to define such constraints. We are therefore not yet fully compliant with R-T4.1-3 and will address this in the next deliverable.

We are mostly compliant with R-T4.1-5, in that we have developed a prototype command line VT that can solve CDL specifications; however, it will need to be expanded as the CDL is expanded to incorporate policies. We are also partially compliant with R-T4.1-6, because the prototype VT can already solve the toy example tasks in seconds; however, we will only be fully compliant when we have carried out a real validation of the tool.

We are fully compliant with R-T4.1-7, as shown in Sections 4.2.2, 4.2.3 and 4.2.4, which define inconsistency specifications for deadlocks, race conditions, and execution loops, respectively. The VT is able to solve specifications involving these inconsistency specifications, and an example of a task for each type of inconsistency is given in the VT repository. Similarly to R-T4.1-6 above, we will only consider ourselves fully compliant with R-T4.1-8 when we have carried out a full validation of the tool, but the VT can solve most of the examples in the repository in seconds. The one task which takes longer is identifying race conditions, but we aim to find a more efficient encoding of this task in future work.

Section 4.2.5 describes how to represent a CDL specification to verify a space of RADON models, showing that we are fully compliant with R-T4.1-11.

6.1 Future work

There are five main aspects of future work:

1. *The representation of policies in the CDL*. Policies are important for analysing the conformance of a RADON model with security requirements. We plan to extend the CDL with a new general policy language. We will also extend the VT in order to enable it to reason about policies encoded in the new language. Finally, we will implement a translation from the new policy language to at least two platform-dependent policy languages (e.g. for AWS and Azure), to enable enforcement of the policies at runtime.
2. *The representation of soft constraints in the CDL*. Currently, when searching for a correction of a RADON model (as described in Section 5.2), the VT finds the *smallest*

correction (i.e. the one with the fewest changes). In the real world, other considerations may be more important; for instance, it may be better to search for the *cheapest* correction, or even the *most secure* correction. We will extend the CDL to allow users to define soft constraints, representing each of these preferences over RADON models.

3. *Validation that the VT runs in real time*. We have defined our validation plans in Deliverable 6.1. We intend to show that on a series of datasets, the VT is capable of solving all tasks in real time.
4. *Learning mode in the VT*. One of the advantages of translating the CDL to ASP in the VT is that there are existing systems for learning ASP [Ray 2009; Corapi et al, 2011; Law et al, 2014, 2015, 2018a]. We aim to extend the verification tool to enable it to learn extra constraints and new policies in a CDL specification, from sets of positive and negative examples. These examples could either be full valid and invalid examples of RADON models, or valid and invalid run-time sequences.
5. *Development of the CDL/VT plugin.* The VT defined in this deliverable is a command line version of the tool. We will develop the CDL/VT plugin, which will call the command line version (running in a container). This plugin will also be able to interact with the Graphical Modelling Tool (GMT) to constrain the set of options a user can select, to those which are consistent with the CDL specification. This will technically require an extra mode of the VT to output the union of all consistent RADON models, but in reality as this functionality is already supported by the underlying ASP solver (known as computing the *brave consequences* of an ASP program), it will only require a single extra flag to be passed to the ASP solver.

References

Brewka, G., Eiter, T., & Truszczyński, M. (2011). Answer set programming at a glance. Communications of the ACM, 54(12), 92-103.

Corapi, D., Russo, A., & Lupu, E. (2011, July). Inductive logic programming in answer set programming. In International Conference on Inductive Logic Programming (pp. 91-97). Springer, Berlin, Heidelberg.

Gelfond, M., & Lifschitz, V. (1988, August). The stable model semantics for logic programming. In ICLP/SLP (Vol. 88, pp. 1070-1080).

Gebser, M., Kaufmann, B., Neumann, A., & Schaub, T. (2007, January). Conflict-driven answer set solving. In IJCAI (Vol. 7, pp. 386-392).

Law, M., Russo, A., & Broda, K. (2014, September). Inductive learning of answer set programs. In European Conference on Logics in Artificial Intelligence (pp. 311-325). Springer, Cham.

Law, M., Russo, A., & Broda, K. (2015). The ILASP system for learning answer set programs.

Law, M., Russo, A., & Broda, K (2018a). Inductive Learning of Answer Set Programs from Noisy Examples. In Advances in Cognitive Systems.

Law, M., Russo, A., & Broda, K. (2018b). The complexity and generality of learning answer set programs. Artificial Intelligence, 259, 110-146.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (TOCL), 7(3), 499-562.

Ray, O. (2009). Nonmonotonic abductive inductive learning. Journal of Applied Logic, 7(3), 329-340.