Petra: Simplified, scalable verification using an object-oriented, compositional process calculus

Extended Abstract

Aran Hakki
University of Southampton
Southampton, UK
ah1m20@soton.ac.uk

Corina Cirstea
University of Southampton
cc2@ecs.soton.ac.uk

Julian Rathke
University of Southampton
Southampton, UK
jr2@ecs.soton.ac.uk

ABSTRACT

Formal methods are yet to be utilized in mainstream software development due to issues in scaling and implementation costs. This work is about developing a scalable, simplified, pragmatic, formal software development method with strong correctness properties and guarantees that are easy prove. The method aims to be easy to learn, use and apply without extensive training and experience in formal methods. Petra is proposed as an object-oriented, process calculus with composable data types and sequential/parallel processes. Petra has a simple denotational semantics, which includes a definition of Correct by Construction. The aim is for Petra is to be standard which can be implemented to execute on various mainstream programming platforms such as Java. Work towards an implementation of Petra as a Java EDSL (Embedded Domain Specific Language) is also discussed.

KEYWORDS

AICC; EDSL, Abstraction; Composition; Correct by Construction; Deterministic; Embedded Domain Specific Language; Parallelism; Refinement; Software Verification; Total Correctness; LaTeX

1 INTRODUCTION

Object-oriented programming (OOP) has been a popular paradigm for many years in mainstream programming. OOP languages such as Java, C#, C++ and Python have consistently occupied top spots in the TIOBE language index [1]. This is largely due to OOP being a natural way for humans to model the real world, hence OOP helps to validate desired behaviour. Validation proves existence of errors in expected behaviour. Verification proves absence of logical errors, which affect expected behavior. Both verification and validation are key in systems engineering. Validation is a human activity and requires domain expertise in order to review designs and create tests. For large systems it is impractical to perform exhaustive checking of expected behaviours, through validation alone. Verification can be automated as it involves checking the logical structure of a system, independent of a particular domain.

Our solution is a language (Petra) which aims for the strongest possible verification checks (whilst being expressive enough for deterministic systems). Petra aims to be easy to program, efficient to check automatically for large systems and easy to validate through both testing and code reviews. This allows developers to spend more time on human activities i.e. coding and validating desired behaviour, with respect to a particular domain. Petra consists of an OO, compositional, process calculus, with an abstract syntax (section 3), a denotational semantics (section 4.2). In addition work has been developed towards a pragmatic implementation of Petra as a Java EDSL. This consists of a concrete Java syntax (section 3.1) and Java implementation of a syntax rewriting system, which gives rise to an operational semantics. The syntax rewriting system rewrites valid Petra programs into programs of sequential transformations only. This simplifies reasoning and automation of reachability checks. In addition this allows for an operational semantics of Petra to be understood as a sequence of transformations on a concrete input state. Please note only the denotational semantics will be discussed in detail. Work towards the rewriting system and implementation can be accessed at [2].

2 METHOD

We prototype a language as an embedded Java DSL, in order to leverage our experience in a mainstream platform and reduce the effort needed to develop tooling such as compilers and IDEs. This means ideas can be tested out quickly, therefore the overall programming experience can be feedback into decision making more frequently. Abstraction, composition and refinement are the key concepts used to develop Petra's programming model.

2.1 State Abstraction & Composition

OOP objects are required to have symbolic abstractions for their underlying state-space. This reduces the cognitive and computational complexity when reasoning about possible states. E.g below we can see below the underlying states space of Pixel is a tuple of 3 bytes i.e. 255 x 255 x 255 values (via the Cartesian product of 3 sets each which represent a byte), however the use-case only requires the pixel to be in either pink or yellow and so we can abstract it using two symbolic states. These capture only these states using predicates over the underlying states space. Only these abstractions are publicly accessible to other objects and this allows the states we are not interested in to be encapsulated.

$$\begin{array}{c} object\ Pixel\{Byte\ R,\ Byte\ G,\ Byte\ B, \\ pink = R\ eq\ 255\ \wedge\ G\ eq\ 192\ \wedge\ B\ eq\ 203, \\ yell = R\ eq\ 255\ \wedge G\ eq\ 255 \wedge B\ eq\ 0\} \end{array} \quad (1)$$

Proc. of the 3rd Agents, Interaction and Complexity Conference (AICC 2022), T. Kelly, A. Masterman (eds.), July 2022, Southampton, UK

This allows us to compose objects and to formally reason about the overall state space. For example consider the composition of two pixels below.

$$\begin{gathered} object\ Pixels\{Pixels\ p1,\ Pixels\ p2, \\ bothPink = p1.pink\ \wedge\ p2.pink, \\ bothYell = p1.yellow\ \wedge\ p2.yellow\} \end{gathered} \tag{2}$$

The overall states pace of this object is given by the Cartesian product of pixels abstracted symbolic states. E.g.

$$\{(pink, pink), (pink, yell), (yell, pink), (yell, yell)\} \tag{3}$$

Then we can resolve the bothPink and bothYellow symbolic states into concrete states by substituting the symbols with sets of symbols recursively using the predicates over the underlying state space, until we arrive at primitive values.

$$\begin{gathered} bothPink = \{((255, 192, 203), (255, 192, 203))\} \\ bothYell = \{((255, 255, 0), (255, 255, 0))\} \end{gathered} \tag{4}$$

2.2 Process Composition

Petra objects can contain steps which are processes either composed of other steps or are primitive steps on primitive data types. Steps are provided with pre/post conditions thus forming a contract (which interpreted as a total function signature) which describes the behaviour of the steps process over a specified state space. The contract is split into one or more kase statements (sub-contracts), each with pre/post-condition pairs, with pre-conditions that represent disjoint states. Steps are composed using these contracts. Currently basic sequential and parallel composition supported (see appendix 3, SEQ_RULE, SEP_SEQ_PAR_RULE), with the aim of supporting various types of parallel and sequential iteration in the future. All computations in Petra are deterministic and terminating thus providing Total Correctness [3]. Each step is interpreted as a total function and both sequential/parallel composition (see SEQ_RULE, SEP_SEQ_PAR_RULE) are defined using operators that ensure only total functions are produced.

2.3 Process Abstraction & Refinement

Each kase sub-contract is an abstraction of the underlying composition of steps within the kase. This abstraction is only valid if the steps composition (also a total function) refines the contract's function (see STEP_RULE and **??**).

2.4 Light System Example

Below we have an example system written in Petra's abstract syntax (Appendix 3). This will be used in simple a proof of refinement using the rules from Petra's denotational semantics (Appendix 4.2) over the abstract syntax. The example will show that the step within the kase statement refines the kase's contract.

2.4.1 *Definitions.*

$$\begin{gathered} object\ Pwr\{\ Bool\ b\ on = t,\ off = f \\ on = kases(kase(on \vee off, on, p.b.setTrue)) \\ off = kases(kase(on \vee off, off, p.b.setFalse))\} \end{gathered} \tag{5}$$

$$\begin{gathered} object\ Btn\{\ Bool\ b\ on = t,\ off = f \\ on = kases(kase(on \vee off, on, b.b.setTrue)) \\ off = kases(kase(on \vee off, off, b.b.setFalse))\} \end{gathered} \tag{6}$$

$$\begin{gathered} object\ Lht\{\ Btn\ b,\ Pwr\ p, \\ on = p.on \wedge b.on, \\ off = p.off \vee b.off, \\ toggle = kases(kase(off, on, Pwr.on \| Btn.on), \\ kase(on, off, Pwr.off; Btn.off))\} \end{gathered} \tag{7}$$

2.5 Example proof: Refinement of 1st kase statement in Light

$$\begin{gathered} Apply\ STEP\_RULE: \\ [\![kase(off, on, Pwr.on \| Btn.on)]\!] = \\ [\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R \ \ if\ [\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R \subseteq\ [\![off, on]\!]^{Lht} \\ where\ R = dom([\![off, on]\!]^{Lht}) \end{gathered} \tag{8}$$

$$\begin{gathered} Apply\ RESOLVE\_CONTRACT\_RULE: \\ [\![off, on]\!]^{Lht} = [\![off]\!]^{Lht} \rightarrow [\![on]\!]^{Lht} = \{(t, f), (f, f), (f, t)\} \rightarrow \{(t, t)\} \end{gathered} \tag{9}$$

$$\begin{gathered} Apply\ ASSUME\_STEP\_RULE: \\ [\![Btn.on]\!] = [\![on \vee off]\!]^{Bnt} \rightarrow [\![on]\!]^{Bnt} = \{t, f\} \rightarrow \{t\} = \{t \mapsto t, f \mapsto t\} \\ [\![Pwr.on]\!] = [\![on \vee off]\!]^{Pwr} \rightarrow [\![on]\!]^{Pwr} = \{t, f\} \rightarrow \{t\} = \{t \mapsto t, f \mapsto t\} \end{gathered} \tag{10}$$

$$\begin{gathered} Apply\ SEP\_SEQ\_PAR\_RULE: \\ [\![Btn.on \| Pwr.on]\!] = [\![Btn.on]\!] \times [\![Pwr.on]\!] \\ [\![Btn.on \| Pwr.on]\!] = \\ \{(t, t) \mapsto (t, t), \\ (t, f) \mapsto (t, t), \\ (f, t) \mapsto (t, t), \\ (f, f) \mapsto (t, t)\} \end{gathered} \tag{11}$$

$$\begin{gathered} [\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R \subseteq\ [\![off, on]\!]^{Lht} \\ where\ R = dom([\![off, on]\!]^{Lht}) = \\ \{(t, t) \mapsto (t, t), \\ (t, f) \mapsto (t, t), \\ (f, t) \mapsto (t, t), \\ (f, f) \mapsto (t, t)\} \\ \subseteq \\ \{(t, f), (f, f), (f, t)\} \rightarrow \{(t, t)\} \\ = \\ \{(t, t) \mapsto (t, t), \\ (t, f) \mapsto (t, t), \\ (f, t) \mapsto (t, t), \\ (f, f) \mapsto (t, t)\} \\ \subseteq \\ \{(t, f) \mapsto (t, t), \\ (f, f) \mapsto (t, t), \\ (f, t) \mapsto (t, t)\} \\ \Longrightarrow \\ [\![kase(off, on, Pwr.on \| Btn.on)]\!] = [\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R = \\ \{(t, f) \mapsto (t, t), \\ (f, t) \mapsto (t, t), \\ (f, f) \mapsto (t, t)\} \\ \blacksquare \end{gathered} \tag{12}$$

3 APPENDIX: SYNTAX

3.1 Abstract Syntax

obj ::= object A $\{\overline{\beta}\ \overline{\gamma}\ \overline{\delta}\}$
$\overline{\beta}$ ::= $B_i\ b_i$
$\overline{\gamma}$ ::= $p_j = e_j$
$\overline{\delta}$ ::= $m_k = ks_k$
e ::= $b_i.p_j \mid e \wedge e \mid e \vee e \mid e + e \mid \neg e$
ks ::= $kases(k_i)$
k ::= $kase(v, w)$
c ::= $p_j \mid e$
v ::= c_p, c_q
s ::= $A.m_k$
w ::= $w; z | z$
z ::= $s \mid parc$
$parc$ ::= $parc \| s \mid s$

3.2 Concrete Syntax (Java)

obj ::= interface A $\{\overline{\beta}\ \overline{\gamma}\ \overline{\delta}\}$
$\overline{\beta}$ ::= $B_i\ b_i;$
$\overline{\gamma}$ ::= $default\ boolean\ p_j\{\ return\ e_j;\}$
$\overline{\delta}$ = $static\ void\ m(A\ a)\{ks;\}$
e ::= $b_i.p_j \mid e\ \&\&\ e \mid e \| e \mid e \hat{} e \mid !e$
ks ::= $kases(a, \overline{k})$
$\overline{k}$ ::= $kase_i(v, a \rightarrow w)$
v ::= p, q
c ::= $a.p_j \mid e$
p ::= $a \rightarrow c_p$
q ::= $a \rightarrow c_q$
w ::= $w\ ;z\ ; \mid z\ ;$
z ::= $seqc \mid join(parc)$
r ::= $a \mid a.b_i$
$seqc$ ::= $seqc\ ;\ seq(r, s) \mid seq(r, s)$
$parc$ ::= $parc\ ,\ par(a \rightarrow a.b_i, s) \mid par(a \rightarrow a.b_i, s)$
s ::= $A :: m$

4 APPENDIX: FORMAL SEMANTICS

A formal semantics of Petra is provided here using a denotational style over an abstract syntax (3) which captures the core concepts and behaviour of the Petra programming language and paradigm. The idea is that Petra programs should only be composed of total functions (transforms) between the observable states of program data. Both the functions and data are stored within objects, similar to objects in OOP. At a high-level the semantics describes an object as a mathematical structure that captures the observable states of the object's data, and the possible transitions between subsets of the observable states. Petra steps are interpreted as total functions, which compose sequentially and in parallel to produce total functions. This simplifies reasoning and programming as total functions are deterministic, terminating and have all their inputs mapped, which means the developer does not have to worry about random behaviours, that can be possibly be non-terminating (which are harder to compose) and about values that which will not be mapped. A description for each part of the syntax will be provided in the next section.

4.1 Explanation of Abstract Syntax

Petra's abstract syntax provides a means of describing objects with state and steps which cause state transitions.

obj, is the entry point into the syntax which provides the structure of an objects definition.

$\overline{\beta}$, is a collection field declarations (indexed over i for use in the denotation semantics that will follow this section).

$\overline{\gamma}$, is a predicate (e) with a label (p). The predicate represents an observable state witin the object, if the predicate evaluates to true then the state described by the predicate is the current state of the object.
$\overline{\delta}$, is a collection of steps each of which are a kases statement (ks), labeled by m (indexed over k for use in the denotation semantics).

e, is a boolean expression language for predicates (where the field terms are indexed by i and the predicate labels are indexed by j, for use in the denotation semantics).

ks, is a kases statement which contains multiple kase statements (k, indexed by i for use in the denotational semantics).

k, is a kase statement which contains the contract (v) and the step composition (w).

c, is a condition used within the kase contract, c can be either a local predicate (p) or a predicate (e).
v, is the kase contract which contains a precondition (c_p) and post-condition (c_q).
s, is a step which is named by the object it is contained within (A) and then by a name for the step within this object (m).
w, is a number of sequentially composed steps (z).
z, is either a named step (s) or a parallel composed step (parc).
parc, is any number of parallel composed named steps (s).

4.2 Denotational Semantics

Definition 4.1. Petra Program.
A petra program is a tuple $< A.m,\ A,\ \overline{obj} >$, where A.m is the root object step (the entry point), A is the root object, $\overline{obj}$ is the set of object definitions.

Definition 4.2. Correct by Construction.
A Petra program is correct by construction if:

$$[\![A.m]\!] : X \rightarrow X$$

$$where\ X = [\![A]\!]_1 \cup \{\bot\}\ and$$
$$[\![A.m]\!]\ is\ a\ total\ function\ with\ the\ codomain\ is\ restricted\ to\ [\![A]\!]_1$$

4.3 Objects

The structure of objects is a tuple containing a set of observable states, a record of predicate labels mapping to subsets of the observable states, and finally a record of step labels which map to total functions.

$$\begin{gathered}[\![A]\!] = (A, [\![\overline{\gamma}]\!], [\![\overline{\delta}]\!]) \\ \text{where each } P_j \subseteq A \\ \text{and each } f_k : A \rightarrow A \\ \\ where \\ [\![\overline{\gamma}]\!] = \{p_j = P_j\} \\ [\![\overline{\delta}]\!] = \{m_k = f_k\}\end{gathered} \tag{13}$$

$$\begin{gathered}[\![object\ A\{\ B_i\ b_i,\ p_j = e_j,\ m_k = ks_k\}]\!] = \\ ([\![B_1]\!]_1 \times [\![B_2]\!]_1 .... \times [\![B_I]\!]_1 \\ , \{p_j = [\![e_j]\!]^A\}, \{m = [\![ks]\!]^A\})\end{gathered} \tag{COMPUTE\_OBJECT\_RULE}$$

The compute objects rule below shows how the set of observable states is the product of all the sets of observable states of the objects fields. Also shown here is how predicate and step labels map to predicate expressions and total function (given by the meaning of kases statements), respectively.

Each primitive type has a corresponding primitive object e.g.

$$\begin{gathered}[\![Bool]\!] = \\ (B, \{isTrue = \{t\}, isFalse = \{f\}\} \\ , \{setTrue = \{t, f\} \rightarrow \{t\},\ setFalse = \{t, f\} \rightarrow \{f\}\})\end{gathered} \tag{ASSUME\_PRIMATIVE\_RULE}$$

The assume primitive rule allows the semantics of primitive objects to be assumed primitive. Note objects are either primitive or non-primitive.

4.4 Predicates

$$[\![b_i.p_j]\!]^A = [\![B_1]\!]_1 \times [\![B_2]\!]_1 \times .... \times ([\![e_j]\!]^A \cap [\![B_i]\!]_1) \times [\![B_{i+1}]\!]_1 \times ... \times [\![B_I]\!]_1$$

$$where\ [\![b_i.p_1]\!]^A \cap [\![b_i.p_2]\!]^A ... \cap [\![b_i.p_J]\!]^A = \emptyset \tag{PREDICATE\_RULE}$$

The predicate rule shows how the terminal term ($b_i.p_j$) of the predicate expression language (e) is interpreted as a product of all

the sets of observable states of the objects fields but with the i^{th} set of observable states effectively filtered (through set intersection) by the set of states represented by the predicate e_j. There is also a constrain that none of the predicates in an object can overlap, i.e. they must be disjoint. This is due to...

$$[\![p_j]\!]^A = [\![e_j]\!]^A \qquad \text{(RESOLVE\_PREDICATE\_RULE)}$$

The resolve predicate rule allows a predicate label to be equated to the set which is constructed by the predicates expression.

The rules below construct sets of observable states using the logical connectives within the predicate expression language, in natural ways using set intersection, union and set difference.

$$[\![e \wedge e']\!]^A = [\![e]\!]^A \cap [\![e']\!]^A \qquad \text{(AND\_RULE)}$$

$$[\![e \vee e']\!]^A = [\![e]\!]^A \cup [\![e']\!]^A \qquad \text{(OR\_RULE)}$$

$$[\![e + e']\!]^A = [\![e]\!]^A \cup [\![e']\!]^A \setminus [\![e]\!]^A \cap [\![e']\!]^A \qquad \text{(XOR\_RULE)}$$

$$[\![\neg e]\!]^A = [\![A]\!]_1 \setminus [\![e]\!]^A \qquad \text{(NOT\_RULE)}$$

*Note that $[\![B_i]\!]_1$ can either be unpacked recursively down to primitive objects, or can be interpreted as the set of predicate labels within B i.e. $[\![B_i]\!]_1 = \{p_j\}$. The former provides a full concrete description of $[\![B_i]\!]_1$. The downside is that the complexity of the product and intersection operations required by COMPUTE_OBJECT_RULE and PREDICATE_RULE would be too large to perform in practice for large systems. However this approach has been taken in the proofs 2.4.1 and 4.6, in order to provide a concrete description of state transitions. The latter allows for a symbolic approach which limits the complexity by only allowing set operations to be computed at the level of abstraction provided by $[\![B_i]\!]$'s predicates labels and the addition of a concretion function would allow the "symbolic" predicate labels to be interpreted as a set of concrete states. This function is provided by the developer through programming the predicates formula, which is built up through layers of sound abstractions from primitive types at the lowest level. The other benefit is that reasoning becomes more explainable as it is performed over names as apposed to fully resolved primitive states.

4.5 Steps

$$[\![A.m]\!] = [\![ks]\!]^A \qquad \text{(RESOLVE\_KASES)}$$

The resolve kases rule allows a step label i.e. a kases statement label, to be equated to the total function constructed by the interpretation of the kases statement.

$$[\![A.m\ ;\ A.m']\!]^A = [\![A.m]\!] \circ [\![A.m']\!] \qquad \text{(SEQ\_RULE)}$$

Sequential step composition is achieved simply through standard function composition of the total functions formed by interpreting the named steps.

$$[\![B_x.m\ ;\ B_{x+c}.m']\!]^A = [\![B_x.m \parallel B_{x+c}.m']\!]^A = z$$

$$where$$

$$z = id_1 \times id_2 ... id_{x-2} \times id_{x-1} \times [\![B_x.m]\!] \times id_{x+1} \times id_{x+2} ... id_{x+c-1} \times [\![B_{x+c}.m']\!] \times id_{x+c+1} \times id_{x+c+2} ... id_{I-2} \times id_{I-1} \times id_I,$$

$$dom([\![B_x.m]\!]^A) \cap dom([\![B_{x+c}.m']\!]^A) = \emptyset\,,$$

$$[\![B_x]\!]_1 * [\![B_{x+c}]\!]_1,$$

$$s, h \models P_1 * P_2 \text{ where there exist } h_1, h_2 \text{ such that } h_1 \perp h_2 \text{ and } h = h_1 \cup h_2 \text{ and } s, h_1 \models P_1 \text{ and } s, h_2 \models P_2,$$

$$where\ *\ is\ a\ separating\ conjunction\ which\ asserts\ that\ the\ heap\ can\ be\ split\ into\ two\ disjoint\ parts\ where\ its\ two\ arguments\ hold.$$

(SEP_SEQ_PAR_RULE)

Separated sequential and parallel composition is achieved through take the product of a sequence of functions. Where the sequence contains a function for each field in the object in which the composition is taking place. For each field the function is either an identity function over the set of observable states in the field or the interpretation of a named step which acts on the field. For this to work the domains of the functions within the product need to be disjoint. Implementations will use separate memory regions to ensure that the observable states will be disjoint and the parallel composed steps can only mutate separate regions hence they cannot interfere with one another. The function product (which can be understood as a generalized version of equation 14, for any number of functions) and there is an identity function id_i for every field not used by any of the steps in the composition. This provides composition and lifting of transforms on separate state spaces, into the state space of the parent object A.

$$\begin{gathered} f : X \to Y,\ g : U \to V \\ f \times g : X \times U \to Y \times V \\ (f \times g)(x, u) = (f(x), g(u)) \end{gathered} \qquad (14)$$

4.6 Kases

$$[\![ks]\!]^A : [\![A]\!]_1 \cup \{\bot\} \rightarrow [\![A]\!]_1 \cup \{\bot\}$$
$$[\![ks]\!]^A = [\![k_1]\!]^A \cup [\![k_2]\!]^A ... [\![k_I]\!]^A \; if$$
$$dom([\![k_1]\!]^A) \cap dom([\![k_2]\!]^A) ... dom([\![k_I]\!]^A) = \emptyset \; and$$
$$[\![k_1]\!]^A, \; [\![k_2]\!]^A, ... \; [\![k_I]\!]^A ... \neq error$$
$$else \; [\![ks]\!]^A = error, \; error = \{\bot \mapsto \bot\}$$

(COMBINE_KASES_RULE)

The combine kases rule provides a kases statement with function interpretation with a signature that is a total transform of the set of observations within the kases object, with the addition of the bottom value, in order to allow for error conditions. The interpretation function definition is the union of all the functions which come from the interpretation of each kase within the kases statement. For this to work the domains of each kase's function must be disjoint, as the union of two functions which map the same inputs does not produce a function.

$$[\![k]\!]^A : [\![A]\!]_1 \cup \{\bot\} \rightarrow [\![A]\!]_1 \cup \{\bot\} \qquad (15)$$
$$[\![k]\!]^A = [\![kase(v, w)]\!]^A$$

The function signature and definition above provides an interpretation of kase statements.

$$[\![kase(v, w)]\!]^A = [\![v]\!]^A \; if \text{ w is a primitive step } \wedge \; [\![v]\!]^A \; is \; a \; total \; function$$

(ASSUME_STEP_RULE)

The assume step rule interprets a primitive kase as the interpretation of its contract, where the contract must be equivalent to a total function.

$$[\![kase(v, w)]\!]^A = [\![w]\!]^A \restriction_R \; if \text{ w is a non-primitive step } \wedge$$
$$[\![w]\!]^A \restriction_R \subseteq \; [\![v]\!]^A$$

$$[\![kase(v, w)]\!]^A = error \; if \text{ w is a non-primitive step } \wedge$$
$$[\![w]\!]^A \restriction_R \not\subseteq \; [\![v]\!]^A$$

$$where \; R = dom([\![v]\!]^A)$$

(STEP_RULE)

The step rule interprets non-primitive kase as the function which results from the interpretation of the kases step composition (w), which has been restricted by the domain of the kases contract, only if the interpreted function fits within the relation provided by the kase contract. Otherwise error returned which is a total function which maps bottom to bottom.

$$[\![v]\!]^A = [\![c_p, c_q]\!]^A = [\![c_p]\!]^A \rightarrow [\![c_q]\!]^A$$
$[\![c_p]\!]^A$ = set of observable input states described by the pre-condition
$[\![c_q]\!]^A$ = set of observable output states described by the post-condition
note v is the kases contract, (see section 3)

(RESOLVE_CONTRACT_RULE)

The resolve contract rule takes the pre/post condition pair and interprets it as a relation between the sets of observable states captured by the pre and post conditions according to the PREDICATE_RULE.

5 APPENDIX: FULL PROOF OF LIGHT SYSTEM

5.0.1 Setup.

$$Apply \; ASSUME\_PRIMATIVE\_RULE :$$
$$[\![Pwr]\!]_1 = \{t, f\} \qquad (16)$$
$$[\![Btn]\!]_1 = \{t, f\}$$

$$Apply \; COMPUTE\_OBJECT\_RULE :$$
$$[\![Lht]\!]_1 = [\![Pwr]\!]_1 \times [\![Btn]\!]_1 = \{(t, t), (t, f), (f, t), (f, f)\} \qquad (17)$$

$$Apply \; RESOLVE\_KASES :$$
$$[\![Lht.toggle]\!]^{Lht} = [\![kases(kase(off, on, Pwr.on \| Btn.on), \qquad (18)$$
$$kase(on, off, Pwr.off; Btn.off))]\!]^{Lht}$$

$$Apply \; COMBINE\_KASES\_RULE :$$
$$[\![kases(kase(off, on, Pwr.on \| Btn.on),$$
$$kase(on, off, Pwr.off; Btn.off))]\!]^{Lht} =$$
$$[\![kase(off, on, Pwr.on \| Btn.on)]\!]^{Lht} \cup \; [\![kase(off, on, Pwr.on; Btn.on)]\!]^{Lht} \qquad (19)$$

5.0.2 Prove Toggle object kase 1.

$$Apply \; STEP\_RULE :$$
$$[\![kase(off, on, Pwr.on \| Btn.on)]\!] =$$
$$[\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R \; if \; [\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R \subseteq \; [\![off, on]\!]^{Lht} \qquad (20)$$
$$where \; R = dom([\![off, on]\!]^{Lht})$$

$$Apply \; RESOLVE\_CONTRACT\_RULE : \qquad (21)$$
$$[\![off, on]\!]^{Lht} = [\![off]\!]^{Lht} \rightarrow [\![on]\!]^{Lht}$$

$$Apply \; RESOLVE\_PREDICATE\_RULE : \qquad (22)$$
$$[\![off]\!]^{Lht} = [\![p.off \vee b.off]\!]^{Lht}$$

$$Apply \; OR\_RULE : \qquad (23)$$
$$[\![p.off \vee b.off]\!]^{Lht} = [\![p.off]\!]^{Lht} \cup [\![b.off]\!]^{Lht}$$

$$Apply \; PREDICATE\_RULE :$$
$$[\![p.off]\!]^{Lht} = \{t, f\} \times \{f\} = \{(t, f), (f, f)\} \qquad (24)$$
$$[\![b.off]\!]^{Lht} = \{f\} \times \{t, f\} = \{(f, t), (f, f)\}$$

$$[\![off]\!]^{Lht} = [\![p.off]\!]^{Lht} \cup [\![b.off]\!]^{Lht} = \{(t, f), (f, f), (f, t)\} \qquad (25)$$

$$Apply \; RESOLVE\_PREDICATE\_RULE : \qquad (26)$$
$$[\![on]\!]^{Lht} = [\![p.on \wedge b.on]\!]^{Lht}$$

$$Apply \; AND\_RULE : \qquad (27)$$
$$[\![p.on \wedge b.on]\!]^{Lht} = [\![p.on]\!]^{Lht} \cap [\![b.on]\!]^{Lht}$$

$$Apply \; PREDICATE\_RULE :$$
$$[\![p.on]\!]^{Lht} = \{t, f\} \times \{t\} = \{(t, t), (f, t)\} \qquad (28)$$
$$[\![b.on]\!]^{Lht} = \{t\} \times \{t, f\} = \{(t, t), (t, f)\}$$

$$[\![on]\!]^{Lht} = [\![p.on]\!]^{Lht} \cap [\![b.on]\!]^{Lht} = \{(t, t)\} \qquad (29)$$

$$[\![off, on]\!]^{Lht} = [\![off]\!]^{Lht} \rightarrow [\![on]\!]^{Lht} = \{(f,t),(f,f),(t,f)\} \rightarrow \{(t,t)\} \tag{30}$$

$$\begin{gathered} \textit{Apply SEP\_SEQ\_PAR\_RULE}: \\ [\![Btn.on \| Pwr.on]\!] = [\![Btn.on]\!] \times [\![Pwr.on]\!] \\ [\![Btn.on \| Pwr.on]\!] = \\ \{(t,t) \mapsto (t,t), \\ (t,f) \mapsto (t,t), \\ (f,t) \mapsto (t,t), \\ (f,f) \mapsto (t,t)\} \end{gathered} \tag{31}$$

$$\begin{gathered} [\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R \subseteq [\![off, on]\!]^{Lht} \\ where\ R = dom([\![off, on]\!]^{Lht}) = \\ \{(t,t) \mapsto (t,t), \\ (t,f) \mapsto (t,t), \\ (f,t) \mapsto (t,t), \\ (f,f) \mapsto (t,t)\} \\ \subseteq \\ \{(t,f),(f,f),(f,t)\} \rightarrow \{(t,t)\} \\ = \\ \{(t,t) \mapsto (t,t), \\ (t,f) \mapsto (t,t), \\ (f,t) \mapsto (t,t), \\ (f,f) \mapsto (t,t)\} \\ \subseteq \\ \{(t,f) \mapsto (t,t), \\ (f,f) \mapsto (t,t), \\ (f,t) \mapsto (t,t)\} \\ \Longrightarrow \\ [\![kase(off, on, Pwr.on \| Btn.on)]\!] = [\![Btn.on \| Pwr.on]\!]^{Lht} \restriction_R = \\ \{(t,f) \mapsto (t,t), \\ (f,t) \mapsto (t,t), \\ (f,f) \mapsto (t,t)\} \\ \blacksquare \end{gathered} \tag{32}$$

5.0.3 *Prove Toggle object kase 2.*

$$\begin{gathered} \textit{Apply STEP\_RULE}: \\ [\![kase(on, off, Pwr.off; Btn.off)]\!] = \\ [\![Btn.off; Pwr.off]\!]^{Lht} \restriction_S \ \ if\ [\![Pwr.off; Btn.off]\!]^{Lht} \restriction_S \subseteq [\![on, off]\!]^{Lht} \\ where\ S = dom([\![on, off]\!]^{Lht}) \end{gathered} \tag{33}$$

$$\begin{gathered} \textit{Apply RESOLVE\_CONTRACT\_RULE}: \\ [\![on, off]\!]^{Lht} = [\![on]\!]^{Lht} \rightarrow [\![off]\!]^{Lht} \end{gathered} \tag{34}$$

$$\begin{gathered} \textit{Apply RESOLVE\_PREDICATE\_RULE}: \\ [\![off]\!]^{Lht} = [\![p.off \vee b.off]\!]^{Lht} \end{gathered} \tag{35}$$

$$\begin{gathered} \textit{Apply OR\_RULE}: \\ [\![p.off \vee b.off]\!]^{Lht} = [\![p.off]\!]^{Lht} \cup [\![b.off]\!]^{Lht} \end{gathered} \tag{36}$$

$$\begin{gathered} \textit{Apply PREDICATE\_RULE}: \\ [\![p.off]\!]^{Lht} = \{t,f\} \times \{f\} = \{(t,f),(f,f)\} \\ [\![b.off]\!]^{Lht} = \{f\} \times \{t,f\} = \{(f,t),(f,f)\} \end{gathered} \tag{37}$$

$$[\![off]\!]^{Lht} = [\![p.off]\!]^{Lht} \cup [\![b.off]\!]^{Lht} = \{(f,t),(f,f),(t,f)\} \tag{38}$$

$$\begin{gathered} \textit{Apply RESOLVE\_PREDICATE\_RULE}: \\ [\![on]\!]^{Lht} = [\![p.on \wedge b.on]\!]^{Lht} \end{gathered} \tag{39}$$

$$\begin{gathered} \textit{Apply AND\_RULE}: \\ [\![p.on \wedge b.on]\!]^{Lht} = [\![p.on]\!]^{Lht} \cap [\![b.on]\!]^{Lht} \end{gathered} \tag{40}$$

$$\begin{gathered} \textit{Apply PREDICATE\_RULE}: \\ [\![p.on]\!]^{Lht} = \{t,f\} \times \{t\} = \{(t,t),(f,t)\} \\ [\![b.on]\!]^{Lht} = \{t\} \times \{t,f\} = \{(t,t),(t,f)\} \end{gathered} \tag{41}$$

$$[\![on]\!]^{Lht} = [\![p.on]\!]^{Lht} \cap [\![b.on]\!]^{Lht} = \{(t,t)\} \tag{42}$$

$$\begin{gathered} \textit{Apply ASSUME\_STEP\_RULE}: \\ [\![Pwr.off]\!]^{Pwr} = [\![on \vee off, off]\!]^{Pwr} = [\![on \vee off]\!]^{Pwr} \rightarrow [\![off]\!]^{Pwr} \\ [\![Btn.off]\!]^{Btn} = [\![on \vee off, off]\!]^{Btn} = [\![on \vee off]\!]^{Btn} \rightarrow [\![off]\!]^{Btn} \end{gathered} \tag{43}$$

$$\begin{gathered} \textit{Apply OR\_RULE}: \\ [\![on \vee off]\!]^{Btn} = [\![on]\!]^{Btn} \cup [\![off]\!]^{Btn} \\ [\![on \vee off]\!]^{Pwr} = [\![on]\!]^{Pwr} \cup [\![off]\!]^{Pwr} \end{gathered} \tag{44}$$

$$\begin{gathered} \textit{Apply PREDICATE\_RULE}: \\ [\![on]\!]^{Btn} = \{t\},\ [\![off]\!]^{Btn} = \{f\} \\ [\![on]\!]^{Pwr} = \{t\},\ [\![off]\!]^{Pwr} = \{f\} \end{gathered} \tag{45}$$

$$\begin{gathered} [\![Btn.off]\!]^{Btn} = \{t,f\} \rightarrow \{f\} = \{t \mapsto f, f \mapsto f\} \\ [\![Pwr.off]\!]^{Pwr} = \{t,f\} \rightarrow \{f\} = \{t \mapsto f, f \mapsto f\} \end{gathered} \tag{46}$$

$$\begin{gathered}
\textit{Apply SEP_SEQ_PAR_RULE}: \\
[\![Btn.off; Pwr.off]\!]^{Lht} = [\![Btn.off]\!] \times [\![Pwr.off]\!] \\
[\![Btn.off; Pwr.off]\!] = \\
\{(t,t) \mapsto (f,f), \\
(t,f) \mapsto (f,f), \\
(f,t) \mapsto (f,f), \\
(f,f) \mapsto (f,f)\}
\end{gathered} \tag{47}$$

$$\begin{gathered}
[\![Btn.off; Pwr.off]\!]^{Lht}\restriction_S \subseteq [\![on, off]\!]^{Lht} = \\
\{(t,t) \mapsto (f,f), \\
(t,f) \mapsto (f,f), \\
(f,t) \mapsto (f,f), \\
(f,f) \mapsto (f,f)\} \\
\subseteq \\
\{(t,t)\} \to \{(t,f),(f,f),(f,t)\} \\
= \\
\{(t,t) \mapsto (f,f), \\
(t,f) \mapsto (f,f), \\
(f,t) \mapsto (f,f), \\
(f,f) \mapsto (f,f)\} \\
\subseteq \\
\{(t,t) \mapsto (t,f), \\
(t,t) \mapsto (f,f), \\
(t,t) \mapsto (f,t)\} \\
\Longrightarrow \\
[\![kase(off, on, Pwr.off; Btn.off)]\!] = [\![Btn.off; Pwr.off]\!]^{Lht}\restriction_S = \\
\{(t,t) \mapsto (f,f)\} \\
\blacksquare
\end{gathered} \tag{48}$$

5.0.4 Proof of root step.

$$\begin{gathered}
32, 48 \implies \\
[\![Lht.toggle]\!] = [\![Btn.on \| Pwr.on]\!]^{Lht}\restriction_R \cup [\![Btn.off; Pwr.off]\!]^{Lht}\restriction_S
\end{gathered} \tag{49}$$

$$\begin{gathered}
[\![Lht.toggle]\!] \\
= \\
[\![Btn.on \| Pwr.on]\!]^{Lht}\restriction_R \cup [\![Btn.off; Pwr.off]\!]^{Lht}\restriction_S \\
= \\
\{(t,f) \mapsto (t,t), \\
(f,t) \mapsto (t,t), \\
(f,f) \mapsto (t,t)\} \\
\cup \\
\{(t,t) \mapsto (f,f)\} \\
= \\
\{(t,f) \mapsto (t,t), \\
(f,t) \mapsto (t,t), \\
(f,f) \mapsto (t,t), \\
(t,t) \mapsto (f,f)\} \\
\Longrightarrow \\
\textit{The light system program is} \\
\textit{Correct by Construction as per definition } 4.2. \\
\blacksquare
\end{gathered} \tag{50}$$

ACKNOWLEDGMENTS

The authors would like to thank Yasser Chaudry (CEO of Sawa Group) and the Chaudry family for their investment in Cognition Box Ltd. This investment allowed for the practical foundations of Petra to be laid by funding early stage research and development (since 2016) and the open-source development of Petra's Java implementation (released in February 2020). Laying foundations is something the family specializes in with their successful construction and engineering firm based in Malawi, Africa. We are very grateful for their help in laying the foundations of our new programming system and believe it will serve a positive impact on society by enabling verifiable, validatable and therefore reliable and explainable software systems at scale.

The work is supported by the EPRSC https://www.ukri.org/councils/epsrc/

REFERENCES

[1] 2022. TIOBE index. https://www.tiobe.com/tiobe-index/. (2022). Accessed: 2022-05-04.
[2] A. Hakki. 2022. (2022). https://github.com/cognitionbox/petra-verification.
[3] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259