RL: a Language for Formal Engineering (Short Paper)

Hadrien Bride
IIIS
Griffith University
Brisbane, Australia
h.bride@griffith.edu.au

Jin Song Dong
School of Computing
Nantional University of Singapore
dcsdjs@nus.edu.sg

Zhé Hóu
School of ICT
Griffith University
Brisbane, Australia
z.hou@griffith.edu.au

Brendan Mahony,
Jim McCarthy
Defence Science and Technology
Department of Defence
Edinburgh, Australia
{Brendan.Mahony, Jim.McCarthy}
@dst.defence.gov.au

***Abstract*—Reflection is a notion that naturally emerges from philosophy, mathematics, and sciences. In short, reflection is the ability of an entity to alter its own behaviour. This paper suggests that reflection is crucial to the development of complex and trustworthy software systems. We present `RL`, a reflective computational model that aims to support the development of a large-scale framework for modelling and manipulating structured data. We give the formal semantics for this computational model. We also share preliminary work on a proof-of-concept implementation of `RL` and discuss future work.**

***Index Terms*—computational model, programming language**

I. Introduction

There are two discernible trends in the development of cyber-physical systems. On the one hand, the complexity of their software and hardware infrastructure is (rapidly) increasing. But, on the other hand, society imposes on them an increasing requirement for verified trustworthiness and such derivative properties as privacy. A primary example of where these trends are creating tension is Defence.

Defence is a large socio-technical system with a critical need for trustworthy information infrastructure that supports all aspects of the Defence enterprise. To date, this need has largely remained unmet; instead, significant applications for Defence enterprise utilise an untrustworthy software stack running on untrustworthy hardware. In this paradigm, despite heroic efforts, it is indeed difficult to assure any practical system security properties that are not simplistic abstractions that fail upon deployment in the 'real world'. This is generally justified by the view that such assurance is impossible, that exploitable functionality is unavoidable and that the infrastructure itself is inherently a contested environment. Much effort has been expended on engaging in the corresponding 'battle-space'. Such attitudes have left Defence networks exposed to attack at the most fundamental level.

One approach to mitigating the hostilities of the network environment is to set up 'islands of trust' with interfaces that offer little or no exploitable functionality: the most successful of which are probably hardware devices that apply information flow control at the boundaries to security zones [1], [5], [19]. The level of simplicity of these devices, and the extent to which they can be transparently implemented in hardware, are the two main factors driving their success in deployment. We feel that the time is right for this thinking to be pushed further into the contested environment, and that key to this will be the introduction of a formal engineering environment tied carefully to a computation model well-suited to being realised in trustworthy hardware – thereby avoiding the complex and untrustworthy system stack.

It is also a good time to take seriously the consequences of a related issue: a large part of the Defence enterprise is devoted to the capability acquisition cycle itself. Defence is therefore fundamentally engaged with the tools and techniques that support capability design, production and through-life maintenance. Of particular interest is the complex body of documentation generated by the Capability Development process [28], including the generation and analysis of the evaluation deliverables – such as Safety and Security Cases – required for accreditation into service. This requires data from a wide variety of activities – system modelling, simulation, active testing to name but a few – to be consistently channeled through a large suite of documents and presented for comprehension by a wide audience of stakeholders.

Early work in this area [8] prototyped a highly-structured language that could be specialised to be domain specific, the elements of which were stored in database tables. This structuring allowed tool support for multi-document consistency through mutually reinforcing mechanisms: syntax-directed editing, with language elements only entered through database references; and, crucially, through calculation on the language structures.

An important aspect highlighted in the above work is reflection, which is the ability of an entity to alter its own behaviour. This ability presupposes the ability to examine its own structure and behaviour. In computer science, computational reflection is the ability of a computational model to examine and modify its own structure and to alter its behaviour at run-time [11], [26], [31].

We assert that reflection is essential for coping with the adaptation and evolution of cyber-physical systems alluded to earlier. Software engineers traditionally perform software evolution through a slow and costly development cycle. In stark contrast with this tradition, modern software programs

are expected to evolve and adapt automatically at run-time. Machine learning and other self-modifying approaches to software development are slowly but surely taking over. An illustrative embodiment of this trend is the fact that even hardware, which is now becoming software thanks to Cloud API, is now expected to adapt dynamically to handle failure and dynamic scaling.

There is also ample evidence that reflective computation may offer significant performance advantages. For instance, there are performance benefits in doing run-time code generation because there is more information available at run-time [22], [27]. It is also beneficial to integrate the consequent interception mechanism with the virtual machine and the just-in-time compiler [16].

Further, arguments for reflection are surprisingly often used in mathematical and common reasoning [3]. Reflection is an important technique in mathematical theorem proving [31] and other formal method approaches [2], [7], [20], [21]. Indeed, fundamentally intuitionstic proof tools such as Coq [4] and Lean [13] derive great utility from extensive use of reflection. It has also been argued that some form of reflection is essential to the development of advanced theorem provers in general [3], [21]. For instance, in [2], it was shown that, in a reflective framework, theorem prover tactics of different forms can be defined, reasoned about (e.g. proved correct) in a formal meta-theory, used to prove theorems interactively and possibly compiled into system code to improve performances.

While reflective code has existed since the earliest days of computing and despite the the previously mentioned benefits, there is ample evidence that it poses clear challenges. We are most concerned that reflective code is generally difficult or impossible to analyse using current tools. The existing literature in the domain of software validation and verification is overwhelmingly focused on the analysis of static programs.

Our work supports the development of a trustworthy and efficient information infrastructure for Defence application. It partcularly focusses on developing a highly-structured and reflective computational environment for the formal engineering of complex hardware and software. In this paper, we present RL – a computational model that is hardware-friendly, highly-structured and reflective. We also present early work based on this computational model: a formal modelling of RL as well as a virtual machine for RL.

II. RL – A Reflective Computational Model

RL (short for ReaL) is a formal computational model designed to be the basis of a large-scale framework for modelling and manipulating structured data. The language is fully reflective: instructions can be manipulated as data and executed at run-time. Its mathematically sound data model makes it an ideal candidate for the modelling of mathematics and scientific knowledge. Its formal semantics enables the construction of well-founded high-level abstractions. Finally, its simplicity and elegance allow it to be effectively implemented both at the software and hardware level depending on the desired use cases.

A. The RL data model

1) Datum: In RL, a *datum* is the most basic piece of data manipulated. An RL datum is a *word* constructed as a sequence of symbols from a finite and totally ordered *alphabet*. The fundamental binary operation on words is *concatenation*, defined in the usual manner. A datum encodes a natural number and arithmetic operations are also well-defined.

Formally, the alphabet of RL is a finite set of symbols denoted Σ together with $\leq_\Sigma$ a total order on Σ. The set of RL words is defined as Σ^* – the Kleene star of Σ (also known as the free monoid on Σ). Further, let $\Sigma = \sigma_1, ..., \sigma_k$ such that $\sigma_i \leq_\Sigma \sigma_{i+1}$ for $1 \leq i < k$. We define $\nu : \Sigma \to \mathbb{N}$, such that $\nu(\sigma_i) = i$ for $1 \leq i \leq k$, the function that maps symbols of Σ to the k first non-zero natural numbers. This numeration of the symbols extends naturally to a numeration of words ν^*, known as the *bijective base-k numeration system* [15], [29].

$$\nu^*(\omega) = \begin{cases} 0 & \text{for } \omega = \epsilon \\ \nu(\omega_0)k^0 + ... + \nu(\omega_n)k^n & \text{for } \omega = \langle \omega_0, \ldots, \omega_n \rangle \end{cases}$$

The numeration ν^* can straightforwardly be used to lift any desired numeric operators onto Σ^*. For instance, the well-ordering on words – known as the shortlex order – can be defined as follows.

$$\omega \leq_{\Sigma^*} \omega' \Leftrightarrow \nu^*(\omega) \leq_{\Sigma^*} \nu^*(\omega') \text{ for } \omega, \omega' \in \Sigma^*$$

Similarly, $+_{\Sigma^*}$ the addition over Σ^* such that $(\Sigma^*, +_{\Sigma^*})$ is isomorphic to $(\mathbb{N}, +)$ can be defined as follows.

$$\nu^*(\omega +_{\Sigma^*} \omega') = \nu^*(\omega) + \nu^*(\omega') \text{ for } \omega, \omega' \in \Sigma^*$$

2) Table: In RL, a collection of labelled datum is called a $table$. A table is an associative map from datum to datum. Formally, a table is a total function of the form $\Sigma^* \to \Sigma^*$.

Tables, as defined in this section, are pervasive throughout computer science. For instance, content-addressable memory (e.g., random access memory) is a form of direct hardware-level support for tables. Tables are commonly referred to as e.g., *dictionaries*, *structures*, *objects*, *records* or *key-value stores*. Elements in the domain of a table are commonly referred to as, e.g., *keys*, *names*, *identifiers*, *addresses*, *attributes* or *properties*, whereas elements in the co-domain of a table are commonly referred to as *values*.

Tables are also the basis for one of the most fundamental concepts in computer science: *indirection*. In computer science, indirection (also called dereferencing) is the ability to indirectly refer to some data by its name, identifier or key. Values of a table can themselves be used as keys.

Note that in practice, although tables are infinite objects, implementation of tables only need to consider a finite number of key-value pairs. By convention, unspecified keys' values are default initialised to zero.

3) Multi-dimensional Table: Multi-dimensional tables are a generalisation of 'tables of tables'. For $n \in \mathbb{N}$, we denote by Ψ^n the set of n-tables – i.e., the set of n-dimensional tables defined as follows.

$$\Psi^n = \begin{cases} \Sigma^* & \text{for } n = 0 \\ \Sigma^* \to \Psi^{n-1} & \text{for } n > 0 \end{cases}$$

Note that, for $i > 1$, an i-table can be converted to an equivalent 1-table via a given bijection[1] $\pi : (\Sigma^*)^i \to \Sigma^*$. Let $\psi \in \Psi^i$ be an i-table; we construct a new function $\psi_\times : (\Sigma^*)^i \to \Sigma^*$ such that $\forall x_1, ..., x_n \in \Sigma^*,\ \psi_\times(\langle x_1, ..., x_n \rangle) = \psi(x_1)...(x_n)$ by uncurrying ψ. Then the required 1-table conversion of ψ is $\overline{\psi} : \Psi^1$ such that for all $x_1, ..., x_n \in \Sigma^*$,

$$\overline{\psi}(\pi(\langle x_1, ..., x_n \rangle)) = \psi_\times(\langle x_1, ..., x_n \rangle) = \psi(x_1)...(x_n)$$

It follows that in theory, multi-dimensional tables may be ignored without loss of generality. Nonetheless, in practice, multi-dimensional tables are a conceptually relevant paradigm well-adapted to user comprehension.

Many programmers are accustomed to working with 2-tables (i.e. contexts) as most programming languages operate on a collection of structures often called *variables*. This is because 2-tables easily leverage indirections to model arbitrary nested data structures, including recursive ones.

Readers may also recognise the traditional relational database data model [9]. Further, note that 2-tables, often viewed as graphs, are used in the domain of knowledge representation [25]. They are commonly referred to as *entity–attribute–value models* or *subject–predicate–object models* (e.g., *semantic triples* [24], *N-Triples* [24]). 3-tables have also been employed in the domain of knowledge representation and are referred to as *context–subject–predicate–object models* (e.g., N-Quads [12]). Note also that the `RL` data model is closely related to the nested table data model – a data-model proposed as a canonical model for data definition and manipulation of forms and form-based documents [23] – as well as other nested relational data models [17].

4) Identifier: An *identifier* is used to designate a particular datum or sub-table of a given multi-dimensional table.

For $0 < j \in \mathbb{N}$, a j-identifier is a j-tuple of datum. We denote by Δ^j the set of j-identifiers (i.e., $\Delta^j = (\Sigma^*)^j$). Then, given an i-table $\psi \in \Psi^i$ with $i \geq j$, $\delta = \langle x_1, ..., x_j \rangle$ is the j-identifier that selects the $(i-j)$-table $\psi(x_1)...(x_j)$.
Notation: in what follows we use the shorthand $\psi[\delta]$ for the table identified by δ in this way.

B. The `RL` Computational Model

In the previous section we described the data model of `RL`. In this section we describe a computational model that operates on this data model. The `RL` computational model is largely inspired from the *random-access machine* computational model [10]. Conceptually, it generalises the *random-access stored-program machine* computational model [14]. As such it is especially well suited for execution on *von Neumann* architectures [30].

A *model* in `RL` is a multi-dimensional table. It evolves in a step-wise process called *execution*. The steps of an execution are discrete *operations* encoded within the model by particular sub-tables called *instructions*. This specialisation to 'model incorporating the operations performed on it' is called an *environment*. The execution of an environment is initiated by the instruction selected by a given identifier called the *entry point*. It may not terminate. In the case of a definitely non-terminating computation there is no computational endpoint.

Fundamental to the computational model is *assignment*, the operation *binding* a value to a particular identifier in a given environment. For $0 < i \in \mathbb{N}$, the set of i-assignments is $\mathscr{A}^i = \Delta^i \times \Sigma^*$. Executing an i-assignment $a = \langle \delta_a, \omega_a \rangle$ in the environment $\psi \in \Psi^i$ results in a new environment $\psi' \in \Psi^i$ such that for all $\delta \in \Delta^i$:

$$\psi'[\delta] = \begin{cases} \omega_a & \text{if } \delta = \delta_a \\ \psi[\delta] & \text{otherwise} \end{cases}$$

Notation: $\psi \xrightarrow{a} \psi'$ denotes that performing the assignment a in the environment ψ results in the environment ψ'.

The remainder of this section introduces the specific instruction set for the `RL` language, stating the role of each instruction and providing its operational semantics. Without loss of generality, we fix $0 < i \in \mathbb{N}$ to be the dimension of environment tables, within which an instruction is encoded by a 1-table (the record identified by a given $(i-1)$-identifier).
Notation: For the operational semantics, let κ be the $(i-1)$-identifier of a given instruction and $\psi, \psi' \in \Psi^i$ be two environment states. Then $\psi \overset{\kappa}{\Rightarrow} \psi'$ denotes that executing κ in the environment ψ leads to a new environment ψ'.

There are two families of instructions in `RL`: *Data instructions*, "immediate" and "indirect", use the assignment operation to modify the environment in which they are executed; *Control-flow instructions* orchestrate the execution of other instructions. Every instruction 1-table record has a non-empty attribute datum, `op`, the value of which we call the corresponding op-code, to distinguish the different members of these families. The remaining attributes of a given instruction record depends on its op-code. The op-codes of the `RL` instruction set are:

- `imMov, imAdd, imConcat` (immediate data instructions);
- `mov, add, concat` (indirect data instructions); and
- `skip, exec, seq, unseq, imBranch, branch` (control-flow instructions).

We now describe each in turn, with the dimension (i), the $(i-1)$-identifier of the instruction (κ), the initial and final environments (ψ and ψ') always related as introduced above.

1) Data Instructions: Data instructions modify a *target* datum within their environment. The i-identifier to this target datum, δ_T, is encoded by record attributes $\mathtt{T}_1, ..., \mathtt{T}_i$; i.e., $\delta_T = \langle \psi[\kappa][\mathtt{T}_1], ..., \psi[\kappa][\mathtt{T}_i] \rangle$.

[1] Such a function is often called a *pairing function* and well-known examples include, e.g., the Cantor pairing function and the Hopcroft-Ullman pairing function.

An *immediate* data instruction modifies the target datum using a value encoded within the instruction itself. Thus, its record has one more attribute, `value`; and,

$\psi \overset{\kappa}{\Rightarrow} \psi' \equiv \psi \xrightarrow{\langle \delta_T, \omega \rangle} \psi'$ where

$$\omega = \begin{cases} \psi[\kappa][\texttt{value}] & \text{for } \psi[\kappa][\texttt{op}] = \texttt{imMov} \\ \psi[\delta_T] +_{\Sigma^*} \psi[\kappa][\texttt{value}] & \text{for } \psi[\kappa][\texttt{op}] = \texttt{imAdd} \\ \psi[\delta_T] \cdot \psi[\kappa][\texttt{value}] & \text{for } \psi[\kappa][\texttt{op}] = \texttt{imConcat} \end{cases}$$

In contrast, an *indirect* data instruction modifies the target datum based on a *source* datum in the environment at large. It has additional attributes $\texttt{S}_1, ..., \texttt{S}_i$ that encode the i-identifier to the source datum $\delta_S = \langle \psi[\kappa][\texttt{S}_1], ..., \psi[\kappa][\texttt{S}_i] \rangle$; and,

$\psi \overset{\kappa}{\Rightarrow} \psi' \equiv \psi \xrightarrow{\langle \delta_T, \omega \rangle} \psi'$ where

$$\omega = \begin{cases} \psi[\delta_S] & \text{for } \psi[\kappa][\texttt{op}] = \texttt{mov} \\ \psi[\delta_T] +_{\Sigma^*} \psi[\delta_S] & \text{for } \psi[\kappa][\texttt{op}] = \texttt{add} \\ \psi[\delta_T] \cdot \psi[\delta_S] & \text{for } \psi[\kappa][\texttt{op}] = \texttt{concat} \end{cases}$$

2) Control-flow Instructions: Control-flow instructions are primarily used to orchestrate data instructions. While data instructions correspond to atomic operational steps (i.e. assignments), control-flow instructions correspond to composite operational steps (i.e. sequences of assignments). The semantics of a control-flow instruction is given with respect to the semantics of the instructions it orchestrates. The latter may never terminate and their computational endpoints may not exist. We abuse notation and use hypothetical computational endpoints when adequate.

3) Skip and Execute: The *skip* instruction (op-code: `skip`) has no additional record attributes and does nothing; i.e.,

$$\psi \overset{\kappa}{\Rightarrow} \psi' \equiv \psi' = \psi$$

The *execute* instruction (op-code: `exec`) transfers control to a specified instruction within the environment and executes it. The $(i-1)$-identifier of the instruction to be executed, κ_I, is encoded as before by the attributes $\texttt{I}_1, ..., \texttt{I}_{i-1}$; and,

$$\psi \overset{\kappa}{\Rightarrow} \psi' \equiv \psi \overset{\kappa_I}{\Longrightarrow} \psi'$$

4) Ordered and unordered sequence: The *ordered sequence* instruction and the *unordered sequence* instruction (op-codes `seq`, `unseq` respectively) execute two instructions; the ordered sequence instruction executes one after the other, whilst the unordered sequence instruction executes them concurrently. The $(i-1)$-identifiers to these two instructions, κ_{I_1} and κ_{I_2}, are encoded as before in the corresponding instruction record by the attributes $\texttt{I}_{1,1}, ..., \texttt{I}_{1,i-1}$ and $\texttt{I}_{2,1}, ..., \texttt{I}_{2,i-1}$, respectively; and,

$\psi \overset{\kappa}{\Rightarrow} \psi' \equiv$

$$\begin{cases} \psi \overset{\kappa_{I_1}}{\Longrightarrow} \overline{\psi} \overset{\kappa_{I_2}}{\Longrightarrow} \psi' \text{ where } \overline{\psi} \in \Psi^i & \text{for } \psi[\kappa][\texttt{op}] = \texttt{seq} \\ \psi \overset{\kappa_{I_1} \otimes \kappa_{I_2}}{\Longrightarrow} \psi' & \text{for } \psi[\kappa][\texttt{op}] = \texttt{unseq} \end{cases}$$

Here $\otimes$ is the *shuffle operator* of shuffle algebra [18], used to denote the arbitrary interleaving of sequences. To understand its role, note that the instructions identified by κ_{I_1} and κ_{I_2} can themselves be sequences of instructions (i.e. the definition of instructions is recursive). Any instruction can be unrolled and viewed as a set of (possibly infinite) sequences of assignments, and it is these sequences that are fed to the shuffle operator. Since assignment is a non-commutative operation, unordered sequence instructions can introduce non-determinism.

5) Branching: The *branching* instructions perform a comparison and, based on the result, execute one of two alternative instructions. Branching instructions introduce the concept of decision – a fundamental concept of computation. We consider two kinds of branching instructions: *immediate* and *indirect*. The distinction is precisely as for data instructions: immediate branching instructions compare a target datum with a value encoded in the instruction itself; indirect branching instructions compare a target datum with a source datum. The data of branching instructions – the target datum, the comparison value or source datum, and the two alternative instructions – are encoded in additional record attributes exactly as described above; for $\psi[\kappa][\texttt{op}] = \texttt{imBranch}$,

$$\psi \overset{\kappa}{\Rightarrow} \psi' \equiv \begin{cases} \psi \overset{\kappa_{I_1}}{\Longrightarrow} \psi' & \text{if } \psi[\delta_T] \leq_{\Sigma^*} \psi[\kappa][\texttt{value}] \\ \psi \overset{\kappa_{I_2}}{\Longrightarrow} \psi' & \text{otherwise} \end{cases}$$

and for $\psi[\kappa][\texttt{op}] = \texttt{branch}$,

$$\psi \overset{\kappa}{\Rightarrow} \psi' \equiv \begin{cases} \psi \overset{\kappa_{I_1}}{\Longrightarrow} \psi' & \text{if } \psi[\delta_T] \leq_{\Sigma^*} \psi[\delta_S] \\ \psi \overset{\kappa_{I_2}}{\Longrightarrow} \psi' & \text{otherwise.} \end{cases}$$

III. Formal Modelling and Implementation

In this section, we briefly introduce some early work based on the `RL` computational model. We present an Isabelle/HOL model of `RL` used to mathematically reason about `RL` models. We also describe a C++ implementation that we use for practical experimentation. The Isabelle/HOL code and the C++ code with examples are available online[2].

A. Isabelle/HOL Modelling

We have modelled a fully expressive subset of the `RL` instructions (`imMov`, `imAdd`, `mov`,`add`, `exec`, `seq`, `branch`) and their semantics in Isabelle/HOL. The other instructions can be modelled using the same technique or simply assembled from existing ones. The Isabelle/HOL code features the modelling of a `RL` model of dimension 2 (2-table). Tables are modelled using the (HOL) map construct. More generally, we make use of tables with mixed rank sub-tables, namely the HOL type: $\Psi^* \equiv \Sigma^* \mid \Sigma^* \rightarrow \Psi^*_\perp$.

The primary goal of this modelling is to check the mathematical consistency of the proposed model in a well-recognised theorem proving framework. The secondary goal is to illustrate how formal models can support engineering. To illustrate the latter, we developed a sorting inference system for `RL`'s data structures in Isabelle/HOL.

In analogy to a typing system, which ensures that functions are applied to arguments in a consistent way, the aim of

[2]https://formal-analysis.com/research/data/iceccs-rl-2020.zip

introducing sortings is to make sure that names, which in RL may refer to tables or fields of tables, are used correctly in a set-theoretic view. We define the following sorts:

$$sort \equiv D \text{ “}\Sigma^* \ set\text{”} \mid T \text{ “}\Sigma^* \to \Sigma^*\text{”} \mid S \text{ “}\Sigma^* \ set\text{”}$$

where D is for datum sort which can be seen as a sort for atomic values and is defined by the set of permitted values. T is for table sort, which defines the sort for tables as a map from a key to the sort of the data which the key refers to. S is for sum sort, which is defined as a set of sub-sorts. Finally, we define an *environment sort* as a map from a sort name to a sort: “$\Sigma^* \to sort$”.

A *sorting sequent* takes the form $\Theta, \psi \vdash t.k : s$ where Θ is an environment sort, ψ is an environment, t is a table, k is a key in t, and s is the sort of the data referred to by k.

$$\frac{\Theta\ s = D\ dms \qquad \psi[t][k] \in dms}{\Theta, \psi \vdash t.k : s}\ datum$$

$$\frac{\Theta\ s = T\ ts \quad \forall k'.ts\ k' \neq \epsilon \Rightarrow S, \psi \vdash (\psi\ t\ k).k' : (ts\ k')}{\Theta, \psi \vdash t.k : s}\ table$$

$$\frac{\Theta\ s = S\ ss \qquad \exists s'.s' \in ss \wedge \Theta, \psi \vdash t.k : s'}{\Theta, \psi \vdash t.k : s}\ sum$$

Figure 1: Sorting inference rules for RL.

The sorting system for RL, given in Figure 1, consists of only three rules, for datum sort, table sort and sum sort respectively. The rule $datum$ states that if the sort s is a datum sort represented by the set dms of permitted values, then the value $\psi[t][k]$ must be in the set dms. The rule $table$ states that if the sort s is a table sort represented by a map ts, then for each key k' such that $ts\ k'$ does not map to ϵ, the sub-table $(t\ k)$ has a key k', which refers to a piece of data of sort $ts\ k'$. Lastly, the rule sum requires that the sort s is a sum sort represented by the set ss of sub-sorts, and there exists a sub-sort $s' \in ss$ such that the key k of table t refers to a piece of data of sort s'.

Notice that the sorting system is independent of the instructions in RL, because instructions are also data in the language.
Example: We define an example environment sort Θ that includes the following mappings:
$\texttt{sVal} \mapsto D\ \{\texttt{0}, \texttt{1}\}$
$\texttt{sKey} \mapsto D\ \{\texttt{Val}\}$
$\texttt{sTab} \mapsto T\ \{\texttt{Val} \mapsto \texttt{sVal}\}$
$\texttt{sInst} \mapsto S\ \{\texttt{exec}, \texttt{imMov}, \cdots\}$
$\texttt{exec} \mapsto T\ \{\texttt{I} \mapsto \texttt{sInst}\}$
$\texttt{imMov} \mapsto T\ \{\texttt{T} \mapsto \texttt{sTab}, \texttt{T}_1 \mapsto \texttt{sKey}, \texttt{Val} \mapsto \texttt{sVal}\}$

In this environment sort, we define a datum sort sVal for atomic Boolean values, a datum sort sKey for a single string value Val, a table sort sTab for simple 1-tables with a single key named Val, a sum sort sInst for instructions, and two table sorts for the two instructions exec and imMov respectively.

We then define an environment ψ where there is a table simpTable1 of sort sTab that has value 0 at the key Val; an instruction instance i1 of imMov that moves the value 1 to the key Val of the target simpTable1; and an instruction instance i2 of exec that executes i1.

Using the rules in Figure 1, it is straightforward to prove in Isabelle/HOL that the two instruction instances and the data structures involved are well-sorted.

B. RLC – Virtual Machine for RL

We have also developed RLC – a C++ implementation of RL which features a efficient virtual machine for an RL model of dimension 2. RLC can load, execute, store and display RL models of dimension 2. The input format is JSON – a language-independent data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs. In general, any structured data format may be used (e.g., XML, S-expression). As first steps towards a high-level input format we introduce the following basic structural constructs:

- *Anonymous indirection* enable users to use arbitrarily nested JSON objects when describing RLC models. During loading, nested JSON objects are flattened using indirections.
- *Namespaces* enable users to hierachically structure their code and use the same identifier to represent different tables in different scopes.
- *Sequences* enable users to use JSON array to model a sequence of ordered instructions.

By convention, RLC restricts attributes' keys to words of the form $[a-zA-Z][a-zA-Z0-9\_:]^*$ in UTF-8 so that RLC models formatted in JSON can be readable using existing text editors. Further, the character : is used as the namespace delimiter that structures the space of keys.

A Datum is implemented as a standard string (i.e. std::string) and the environment is implemented using standard maps (i.e. std::map). For efficiency and to match the standard string implementation, symbols of the alphabet are encoded on a single byte. It follows that, in RLC, there are 2^8 symbols. For demonstration purposes, we use the standard C++ map implementation – a well-tested and robust associative array implemented as red-black trees. Note that, it is straightforward to replace it by other key-value store implementations (e.g., hashmap, graph database).

The execution of the RL instruction set (i.e., imMov, imAdd, inConcat, mov, add, concat, skip, exec, seq, unseq, imBranch, branch) is implemented using a recursive C++ function. This implies that, during execution, instructions left to be executed are stored in the call stack, which is hardware-specific nowadays. In addition to basic instructions, RLC introduces a new instruction called the clear instruction (op-code: clear) which encodes an identifier and when executed clears it. This instruction is introduced for convenience purposes and helps to manage memory.

The purpose of RLC is to bootstrap the development of high-level computations in RL, especially to develop high-level language constructs in RL. These include well-known control-flow structures, allocator, and stacks. To illustrate how a type system may be implemented in RLC, we defined and implemented handy type system that constrains the structure of RLC tables. Additionally, to demonstrate RL's ability to manipulate structured data, we notably implemented a propositional logic framework in which propositional expressions can be represented and manipulated.

IV. Conclusion and Future Work

In this paper, we presented our line of research which aims to develop transparent and trustworthy information infrastructure for Defence applications. This work is focused on the hardware/software transition layer. Specifically, we proposed RL – a computational model that is hardware-friendly, highly-structured and reflective. We formally described its data model and operational semantics. As practical contributions, RL structure and semantics have been formally modelled in Isabelle/HOL to ensure they are both mathematically sound. Further, we also developed a proof-of-concept implementation with which we demonstrated high-level computation features such as a type-system and a propositional logic reduction system. These contributions are made public and, we hope, will foster the development of a comprehensive framework for formal engineering of high-level and large-scale information infrastructure.

In the future, we seek to build upon RL and develop a full-fledged interactive development environment to provide high-level support for large-scale knowledge management and capability development tasks. RL is well suited for modelling mathematics and scientific knowledge. Its semantics is formally defined and its mathematical foundations well-established. It also supports reflection, a notion that appears surprisingly often used in mathematical and common reasoning. We also plan to implement, within RL, formal analysis tools that support RL developments.

Finally, we will explore the design of dedicated hardware that directly supports our reflective computational model. Indeed, in contrast with other reflective languages which follow a functional approch to computation, RL adopts a structural approch to computation [6] which is well suited for implementations on *von Neumann* architectures [30].

References

[1] M Anderson, C North, J Griffin, R Milner, J Yesberg, and K Yiu. Starlight: Interactive link. In *Computer Security Applications Conference, Annual*, pages 55–55, 1996.

[2] Alessandro Armando, Alessandro Cimatti, and Luca Viganò. Building and executing proof strategies in a formal metatheory. In *Congress of the Italian Association for Artificial Intelligence*, pages 11–22. Springer, 1993.

[3] Sergei N Artëmov. On explicit reflection in theorem proving and formal verification. In *International Conference on Automated Deduction*, pages 267–281. Springer, 1999.

[4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The coq proof assistant reference manual. *INRIA, version*, 6(11), 1999.

[5] Mark Beaumont, Jim McCarthy, and Toby Murray. The cross domain desktop compositor: Using hardware-based video compositing for a multi-level secure user interface. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 533–545, 2016.

[6] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[7] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *International Symposium on Theoretical Aspects of Computer Software*, pages 515–529. Springer, 1997.

[8] Tony Cant, Ben Long, Jim McCarthy, Brendan Mahony, and Kylie Williams. The HiVe writer. *Electronic Notes in Theoretical Computer Science*, 217:221–234, 2008.

[9] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[10] Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[11] Stefania Costantini. Meta-reasoning: a survey. In *Computational Logic: Logic Programming and Beyond*, pages 253–288. Springer, 2002.

[12] Richard Cyganiak, Andreas Harth, and Aidan Hogan. N-quads: Extending n-triples with context. *W3C Recommendation*, page 41, 2008.

[13] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[14] Calvin C Elgot and Abraham Robinson. Random-access stored-program machines, an approach to programming languages. In *Selected Papers*, pages 17–51. Springer, 1982.

[15] James E Foster. A number system without a zero-symbol. *Mathematics Magazine*, 21(1):39–41, 1947.

[16] Michael Golm and Jürgen Kleinöder. Jumping to the meta level. In *International Conference on Metalevel Architectures and Reflection*, pages 22–39. Springer, 1999.

[17] Alexander Gorelik, Sachinder Chawla, Awez Syed, Leon Burda, Mon Yee, and Sridhar Grantimahapatruni. Nested relational data model, November 29 2001. US Patent App. 09/782,186.

[18] J.A. Green. *Shuffle algebras, lie algebras and quantum groups*. Textos de matemática // Departamento de Matemática, Faculdade de Ciências e Tecnologia, Universidade de Coimbra. Departamento de Matemática da Universidade de Coimbra, 1995.

[19] Duncan A Grove, Toby C Murray, Chris A Owen, Chris J North, JA Jones, Mark R Beaumont, and Bradley D Hopkins. An overview of the annex system. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 341–352. IEEE, 2007.

[20] Jin Song Dong Rajeev Gore Zhe Hou Brendan Mahony Hadrien Bride, Cheng-Hao Cai and Jim McCarthy. N-pat: A nested model-checker (system description). In *International Joint Conference on Automated Reasoning, 2020*, 2020.

[21] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical report, Citeseer, 1995.

[22] David Keppel, Susan J Eggers, and Robert R Henry. *A case for runtime code generation*. Department of Computer Science and Engineering, University of Washington, 1991.

[23] Hiroyuki Kitagawa and Tosiyasu L Kunii. Nested table data model (ntd). In *The Unnormalized Relational Data Model*, pages 17–68. Springer, 1989.

[24] Ora Lassila, Ralph R Swick, et al. Resource description framework (rdf) model and syntax specification. 1998.

[25] Hector J Levesque. Knowledge representation and reasoning. *Annual review of computer science*, 1(1):255–287, 1986.

[26] Pattie Maes and Daniele Nardi. Meta-level architectures and reflection. 1988.

[27] Henry Massalin. Synthesis: An efficient implementation of fundamental operating system services. 1993.

[28] Commonwealth of Australia. Defence capability development handbook. 2012.

[29] Raymond M Smullyan et al. *Theory of formal systems*. Princeton University Press, 1961.

[30] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[31] Ming-Yuan Zhu. Computational reflection in powerepsilon. *SIGPLAN Notices*, 29(1):13–19, 1994.