L4 Language Report

October 28, 2022

L4 Development Team

https://cclaw.smu.edu.sg/

Contents

List of Figures

1 User Documentation

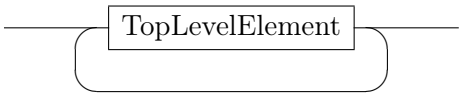
1.1 Introduction

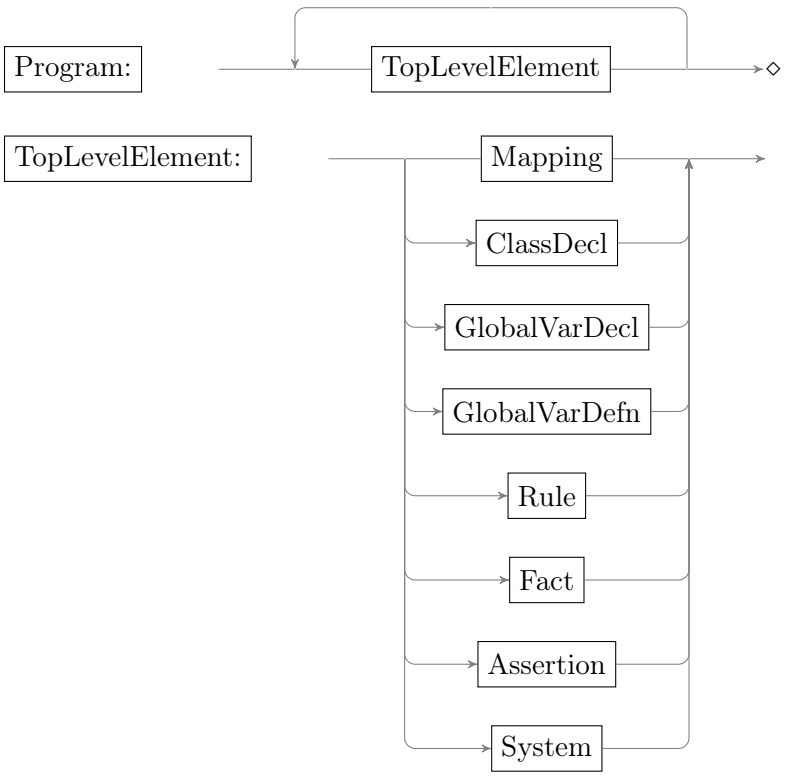
Describing the global idea of L4 — MS: TBD

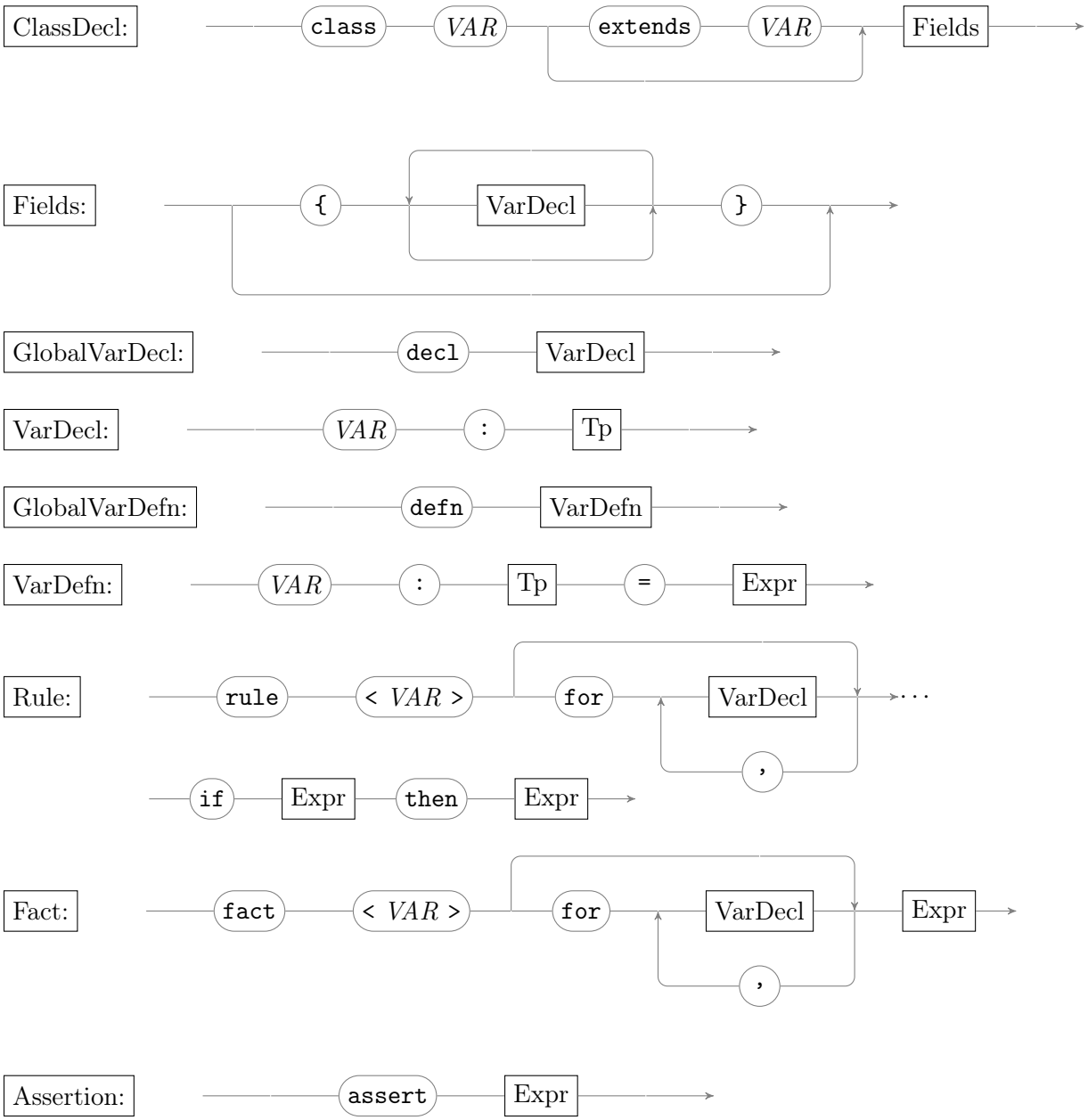
1.2 L4 Language Overview

1.2.1 Syntax

Program







A Program is the main processing unit. It consists of a list of top level elements that can be arranged in any order. A TopLevelElement can be a mapping, a list class declarations, global variable declaration or definition, a rule, fact or assertion of a system.

IS: Inari: more etails

A Mapping maps identifiers of the program to GF strings.

Class declarations (ClassDecl) come in several shapes. In its simplest form, a class declaration introduces a new class and relates it to a superclass, as in

```
class Lawyer extends Person
```

The `extends` clause can also be omitted, in which case the superclass is implicitly assumed to be `Class` (see Section 1.2.2 for the built-in class hierarchy). Thus,

```
class Person
```

is equivalent to:

```
class Person extends Class
```

New fields can be added to a class with field declarations (Fields). These are variable declarations enclosed in braces; they can also be missing altogether (equivalent to empty field declarations `{}`). For example,

```
class Person {
   name : String
   age : Integer }

class Lawyer extends Person {
   companyName : String }
```

introduces two fields `name` and `age` to class `Person`, whereas `Lawyer` inherits `name` and `age` from `Person` and in addition defines a third field `companyName`.

Global variable declarations (GlobalVarDecl) introduce names that are meant to have global visibility in the program.

A Rule introduces local variables in a `for` clause (which may be omitted if there are no local variables), followed by a precondition (`if`) and a postcondition (`then`), both assumed to be Boolean expressions. If there are several preconditions, these have to be conjoined by *and* to form a single formula. A Fact is a notational variant of a Rule that does not have a precondition (more technically, a fact is mapped to a rule whose precondition is `True`).

An Assertion is a Boolean expressions introduced by the keyword `assert`.

1.2.2 Typing

1.2.3 Pragmatics

1.3 Specific Language Elements

1.3.1 Automata

Automata are an appropriate means for describing *time-dependent* systems, as opposed to rules and facts that provide a *static* view.

In the present form, the L4 automata are inspired by the theory of Timed Automata [LPY97] and in particular by the Uppaal systen[1]. Our long-term aim is to go beyond the model checking capabilities of Uppaal by a mapping of automata to SMT solvers.

[1] https://uppaal.org/

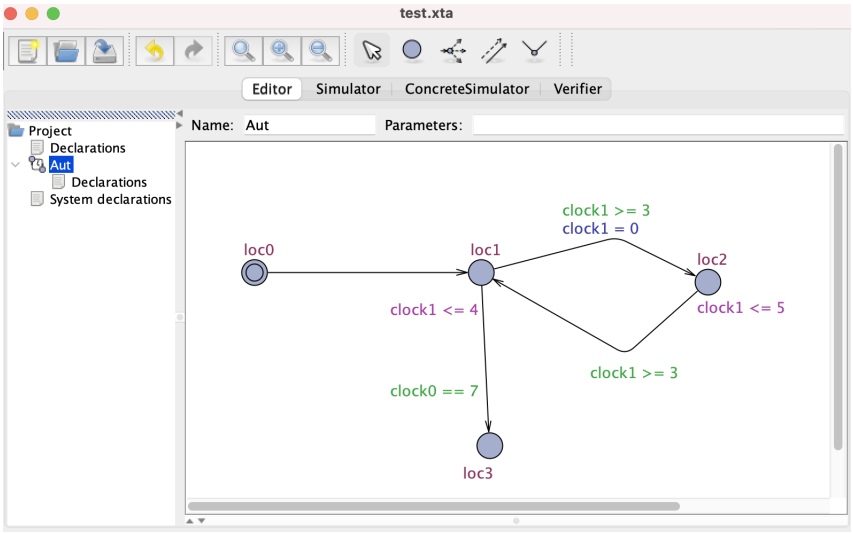


Figure 1.1: Automaton represented in Uppaal

As long as this feature is under development, we try to achieve maximal compatibility so that Uppaal can be used together with L4, in two directions:

- The internal L4 data structures can be printed in the Uppaal `*.xta` format. An example is given in Figure 1.2.
- `*.xta` files written by Uppaal can be read by L4, with minor syntactic differences and restrictions. An example is given in Figure 1.3.

The above automata are graphically displayed in Figure 1.1.

These two methods will be revisited in the following.

1.3.1.1 Automata and Systems

An *automaton* is a graph commposed of nodes (states) and arcs (transitions). States and transitions can be annotated with conditions (temporal or not) and synchronization information. A *system* is composed of one or several automata and may contain declarations of channels (for synchronization), clocks etc.

1.3.1.2 From L4 to Uppaal

The Uppaal `*.xta` format contains all the relevant structural information for processing an automaton in Uppaal. However, no graphical information is available, which typically leads to a poor layout when opening an `*.xta` file.

An L4 program containing a system can be printed in Uppaal syntax with the fake assertion

```
assert {printUp} True
```

```
chan x, y;
process Aut () {
    clock clock0, clock1;
    state
        loc0,
        loc1 { clock1 <= 4 },
        loc2 { clock1 <= 5 },
        loc3
        ;

    init loc0;
    trans
        loc1 -> loc3 { guard clock0 == 7 ;   },
        loc2 -> loc1 { guard clock1 >= 3 ;   },
        loc1 -> loc2 { guard clock1 >= 3 ;  assign clock1 = 0; },
        loc0 -> loc1 {    }
        ;
    }
system Aut;
```

Figure 1.2: System specification in Uppaal format

```
system AutSys {
    chan x, y;
    process Aut () {
        clock clock0, clock1;
        state
            loc0,
            loc1 { clock1 <= 4 },
            loc2 { clock1 <= 5 },
            loc3
            ;

        init loc0;
        trans
            loc1 -> loc3 { guard clock0 == 7 ;   },
            loc2 -> loc1 { guard clock1 >= 3 ;   },
            loc1 -> loc2 { guard clock1 >= 3 ;  assign clock1 = 0; },
            loc0 -> loc1 {    }
            ;
        }
    }
```

Figure 1.3: System specification in L4 format

which can be run as described in Section 1.3.3.2. The resulting text can be written on a `*.xta` file and be processed by Uppaal. The directive `printL4` produces the L4 format of automata.

Genuine proof obligations are currently not written to Uppaal (they would have to be written to the corresponding `*.q` file).

1.3.1.3 From Uppaal to L4

A Uppaal `*.xta` file can be copied into L4 and then be manipulated as any other L4 source. The L4 syntax and Uppaal syntax are sufficiently similar so as to not require major changes, but the following manual postprocessing is necessary:

- Automata and declarations belonging to one system have to be included into a system specification `system SysName { ... }` as in Figure 1.3. Several system specifications may coexist in an L4 file.
- Comments begin with a double slash (`//`) in Uppaal, which is not recognized by the L4 parser, and therefore have to be replaced by a sharp (`#`) in L4.
- Some expressions may be written differently, in particular the constants `True` and `False` which are written lowercase in Uppaal.

A notable difference is that Uppaal permits the declaration of automaton *templates* that can be instantiated. This is currently not possible in L4.

MS: BEGIN TODO

Note: there is another TODO file elsewhere. Integrate in this document, in a separate file to be excluded from compilation for the general public.

- local variable declarations not printed

MS: END TODO

1.3.2 Rules

1.3.3 SMT Solver

1.3.3.1 Assertions

1.3.3.2 Running the solver

1.3.4 Expert Systems

Expert Systems are a way of replicating the logical reasoning and domain capabilities of a human expert through explicit knowledge capture.

L4 targets the use of expert systems as formal advisory interpreters; that form high-level conclusions given a set of ground truths and the application of formal axioms.

LF: There ex-
ts case-based
gal expert sys-
ems, but my
ersonal opinion
that these are
ss intuitive and
uch harder to
upport

Legal texts are typically well-structured & formal, with well-defined terms integrated in clear and specific domains. However, there might be situations where the interpretive ability of a trained lawyer are required, but cannot be accessed due to the insufficient time or resources. Situations like these are prime areas that expert systems excel in. Unfortunately, previous attempts at legal-specific expert systems in the 1980s and 1990s were largely exploratory and descriptive in nature, with no attempts at public testing. [Lei16]

In the time since, there have been various efforts into the building of rigorous and efficient rule-based systems. To build on the results of these efforts, we offer L4 as an upstream format that transpiles to such existing formats.

ALF: To cite drool's rete, find clara's paper (if it exists) etc.

As with existing expert systems, L4 files already mimic the characteristics of an easily modifiable knowledge base. Furthermore, this approach allows you, the user-developer/knowledge engineer, the option of choosing the language environment you would like to work with, and the flexibility of desigining a frontend user interface that would fit your purposes.

The currently supported rule-engine output formats are Drools and Clara-Rules.

2 Developer Documentation

2.1 Parsing and Typechecking

Describes the parsing technlogy and typechecking procedures

2.2 Translations to Proof and Reasoning Tools

2.2.1 Translation – Overview

2.2.2 Translation to SMT solvers

2.2.3 Translation to ASP solvers

We are currently working on the use of Answer Set Programming (ASP) for various functionalities. Our main choice of ASP solver is Clingo. Clingo is our choice of solver as it is a robust implementation of ASP and also allows additional constructs like *choice rules* and *weak constraints* (ie optimizations) with *priority levels*

2.2.3.1 ASP for defeasible reasoning

Defeasible reasoning can be modelled in ASP through use of the *negation-as-failure* operator *not* which is interpreted under the *stable model semantics*. Our guiding principle is that it should be possible to simulate various kinds of defeasible/non-monotonic logics by principled translations to the correct corresponding ASP program where we there is only one non-monotonic operator namely, the negation-as-failure *not* as described above. Simulating everything on ASP has the advantage that we can use the underlying stable model semantics to give clear axiomatic foundations for the overall semantics of the defeasible reasoning processes that we implement. A theoretical investigation into the limits of a such a 'reductionist' approach to defeasible reasoning would be of interest. We shall now describe one implementation of defeasible reasoning for which we currently have a rudimentary L4 to ASP code generator.

2.2.3.2 Assumed input rule syntax

We assume that the input rule syntax from a source such as L4 is compatible with the syntax for prolog-like rules. More specifically we assume that each source rule has exactly the following form (or can be put into the following form):

```
pre_con_1(V1),pre_con_2(V2)...,pre_con_n(Vn) -> post_con(V).
```

We further make the folllowing assumptions:
1. Each pre-condition $pre\_con_i(V_i)$ is atomic and so is the post-condition $post\_con(V)$.
2. V_i is the set of variables occuring in the i^{th} pre-condition $pre\_con_i(V_i)$ and V is the set of variables occuring in the post condition $post\_con(V)$. We assume that $V_1 \cup V_2 \cup ... \cup V_n = V$.
3. Each variable occurring in either a pre-condition or the post condition is universally quantified over.
4. Each input rule of the form above carries with it an integer rule id. possibly originating in the legislation.

2.2.3.3 Main input rule translation

Given an input rule

```
pre_con_1(V1),pre_con_2(V2)...,pre_con_n(Vn) -> post_con(V).
```

obeying all the properties in (2), say this rule has integer rule id. n, then we write the following rule in our ASP program:

```
according_to(n,post_con(V)):-legally_holds(pre_con(V1)),...,legally_holds(pre_con(Vn)).
```

We repeat this for each input rule.

2.2.3.4 Defeasibility meta-theory

The following set of meta-rules governs the defeasibility in our forward reasoning process (This is only one possible implementation of defeasibility):

```
defeated(R2,C2):-overrides(R1,R2),according_to(R2,C2),legally_enforces(R1,C1),
opposes(C1,C2).

opposes(C1,C2):-opposes(C2,C1).

legally_enforces(R,C):-according_to(R,C),not defeated(R,C).
```

```
legally_holds(C):-legally_enforces(R,C).

:-opposes(C1,C2),legally_holds(C1),legally_holds(C2).
```

Given any atomic propositions C, $C1$ such that C and $C1$ are negatives of each other we add the ASP rule:

```
opposes(C,C1):-according_to(R,C).
opposes(C1,C):-according_to(R1,C1).
```

2.2.3.5 ASP for Proof Search/Abductive reasoning

Implementing a proof search procedure in ASP has multiple potential uses such as solving abductive reasoning problems, general model checking and automating/optimizing the generation of user questions to determine the truth value of a base query. The procedure presented here is a goal-directed procedure that combines forward chaining and backward chaining and makes use of the choice rules and prioritized weak constraints features of clingo. Unlike many papers on abductive reasoning, the space of abducibles for our abduction problems is not pre-set and must be derived from the rules itself.
The intuitive idea that will enable us to build a theorem prover is to generate the space of abducibles in a controlled manner by applying the top level L4 input rules in the reverse direction. Ie recursively inferring pre-conditions from post conditions, and then feeding these abducibles as initial conditions into the forward reasoning mechanism to check for entailment of the base query. This reverse inference procedure for generating the space of abducibles is similar to Prolog's resolution and unification algorithms. There are technical issues to be dealt with here regarding the handling of certain existential quantifiers via skolemnization, the completeness of the search procedure and the handling of arithmetic operations/constraints. These issues will be described later. First we shall describe the abduction space generation procedure for our restricted rule syntax.
Given an input rule

```
pre_con_1(V1),pre_con_2(V2)...,pre_con_n(Vn) -> post_con(V).
```

obeying all the properties in (2), we add the following set of ASP rules to our ASP program:

```
explains(pre_con_1(V1),post_con(V),N+1):-query(post_con(V),N).
explains(pre_con_2(V2),post_con(V),N+1):-query(post_con(V),N).
                              .
                              .
                              .

explains(pre_con_n(Vn),post_con(V),N+1):-query(post_con(V),N).
```

We repeat this for each input rule. We then add the following bit of ASP code:

2.2.3.6 Supporting code

```
query(C,0):-generate_q(C).
query(C1,N):-query(C,N),opposes(C,C1).

query(X,N):-explains(X,Y,N),q_level(N).
```

For each pair of atomic propositions C, $C1$ that are negatives of each other, we add the following ASP rule:

```
opposes(C,C1):-query(C,N).
opposes(C1,C):-query(C1,N).
```

2.2.3.7 ASP for Justification tree generation

The method for deriving justification trees is similar to that of proof search in that we seek to recursively justify premises of justified conclusions. However as this is a trace extraction problem rather that a proof search problem, this method is in fact easily extended as is to rules involving existential quantifiers in antecedents, arithmetic inequalities etc. Given an input rule

```
pre_con_1(V1),pre_con_2(V2)...,pre_con_n(Vn) -> post_con(V).
```

obeying all the properties in (2), with integer rule id n we add the following set of ASP rules to our ASP program:

```
caused_by(pos,legally_holds(pre_con_1(V1)),according_to(n,post_con(V)),N+1)
:-according_to(n,post_con(V)),legally_holds(pre_con_1(V1)),...,legally_holds(pre_con_n(Vn)
justify(according_to(n,post_con(V),N).
            .
            .
            .
caused_by(pos,legally_holds(pre_con_n(Vn)),according_to(n,post_con(V)),N+1)
:-according_to(post_con(V)),legally_holds(pre_con_1(V1)),...,legally_holds(pre_con_n(Vn)),
justify(according_to(n,post_con(V),N).
```

We repeat this for each input rule and then add the following bit of ASP code

2.2.3.8 Supporting code

```
caused_by(pos,overrides(R1,R2),defeated(R2,C2),N+1):-defeated(R2,C2),overrides(R1,R2),
according_to(R2,C2),legally_enforces(R1,C1),opposes(C1,C2),justify(defeated(R2,C2),N).

caused_by(pos,according_to(R2,C2),defeated(R2,C2),N+1):-defeated(R2,C2),overrides(R1,R2),
according_to(R2,C2),legally_enforces(R1,C1),opposes(C1,C2),justify(defeated(R2,C2),N).

caused_by(pos,legally_enforces(R1,C1),defeated(R2,C2),N+1):-defeated(R2,C2),overrides(R1,R
according_to(R2,C2),legally_enforces(R1,C1),opposes(C1,C2),justify(defeated(R2,C2),N).

caused_by(pos,opposes(C1,C2),defeated(R2,C2),N+1):-defeated(R2,C2),overrides(R1,R2),
according_to(R2,C2),legally_enforces(R1,C1),opposes(C1,C2),justify(defeated(R2,C2),N).


caused_by(pos,according_to(R,C),legally_enforces(R,C),N+1):-legally_enforces(R,C),accordin
not defeated(R,C),justify(legally_enforces(R,C),N).

caused_by(neg,defeated(R,C),legally_enforces(R,C),N+1):-legally_enforces(R,C),according_to
not defeated(R,C),justify(legally_enforces(R,C),N).


caused_by(pos,legally_enforces(R,C),legally_holds(C),N+1):-legally_holds(C),legally_enforc
not user_input(pos,C), justify(legally_holds(C),N).

caused_by(pos,user_input(pos,C),legally_holds(C),N+1):-legally_holds(C),
user_input(pos,C), justify(legally_holds(C),N).

justify(X,N):-caused_by(pos,X,Y,N),graph_level(N+1), not user_input(pos,X).
directedEdge(Sgn,X,Y):-caused_by(Sgn,X,Y,M).

graph_level(0..N):-max_graph_level(N).

justify(X,0):-gen_graph(X).
```

2.2.3.9 General discussion

As alluded to earlier, further work remains to be done especially with the proof search procedure. Dealing with input rules that have existential quantifiers in antecedents means we have to introduce *skolem* functions and variables when reversing the rules for the generation of abducibles. Further modifications are necessary to generate the appropriate abducibles based on partial substitution of variables. Ie. When one conjunct

in a rule premise has been instantiated by the user with some variable substitutions then the other conjuncts should be instantiated with the same substitutions and the resulting set of atoms made available as possible abducibles in the abductive reasoning process. This to some extent bridges the gap between Prolog, whose aim is to find some proof of the query by among other things making appropriate substitutions and ASP, whose aim is to give the set of derived consequences based on ground facts and first order rules with no real notion of a 'query'.
Furthermore the methods presented here can give rather unwieldy outputs with rules that have heavy use of arithmetic constraints, along with existential quantifiers in rule premises etc. This is the case for example with temporal rules that describe time-dependent regulative norms. Essentially such rules are not easily 'reversible' although other model checking, contract execution functionalities that do not depend on the reversibility of rules can still be carried out. For model checking of such rules however it is likely that a SMT solver approach is the best approach. A contract execution (as opposed to a contract verification) language however can be realised in ASP by adapting the use of event calculus formalisms and such a language can be enriched with some lightweight contract checking features.
Finally of course it remains to prove the completeness of the proof search procedures. It is expected that such completeness results will be closely tied to the choice of defeasibility meta-theory that underlies the forward reasoning process.

2.2.4 Translation to Expert Systems

3 Internal

Attention, this chapter should be commented out for publication of the document

3.1 TODO

The following issues are still unresolved

3.2 Collection of thesis / internship topics

Collection of ideas for subprojects with CCLAW

3.3 UPPAAL-style timed automata based semantics for Natural L4

- Each natural l4 rule gives rise to a fragment of an automaton.
- In the end, we stitch all these partial automata together.
- Diff rules may talk about the same automata, need to figure out how to sync and connect them together.

3.3.1 Natural l4 to automata

- § `Rule name`

  this defines a state which is the entry point to the rest of the automaton representing the rule

- `Every/Party (entity label)`

  This indicates that the rule to be constructed is part of the automaton corresponding to the entity label.

  - Q: Any semantic diff between every and party?

- `subject constraint`

  not sure what this means

- `attribute constraint`

  not sure about this either

- `conditional constraint`

  ie if, when, unless

  These are guards on the action, represented as a transition

  Need to clarify: when = if, unless = if not

- `upon trigger`

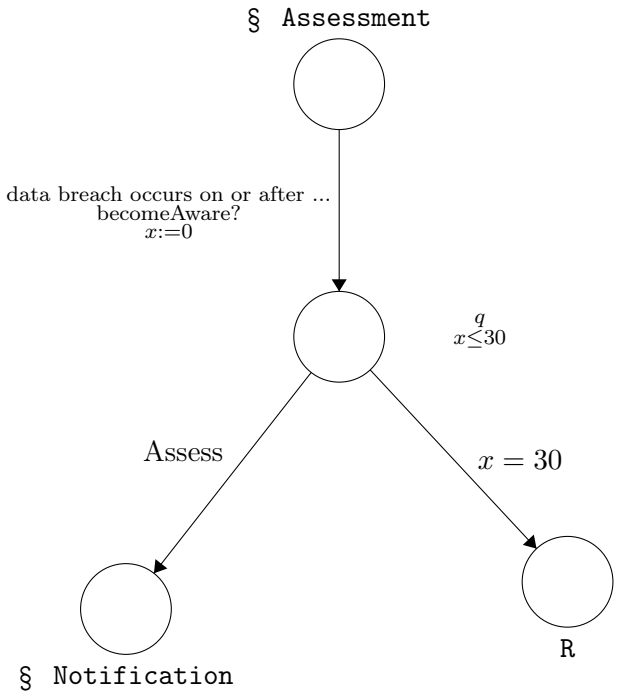
  upon action a creates a transition labelled with the action a into this automaton.

  synchronize with other action a somewhere else in the network.

- How to handle clocks? What clock variables to use and when to reset?
- switch/case or match would be nice

  find some concrete example for motivation

As an example, consider the § `Assessment` rule in the PDPA case. This gives rise to an automaton with a structure as shown below.



Note here that:

- For simplicity, we ignore attribute constraints and subject constraints like `WHO`, `WHICH`, etc.
- "data breach occurs on or after ..." is the conditional constraint, represented by a guard on the transition leading into the rest of the rule.
- "becomeAware" is a synchronization action, corresponding to the "`UPON` becoming aware..." part.
- x is taken to be a fresh clock variable.
- q is a fresh, anonymous state whose main purpose is to hold the clock invariant of $x \leq 30$.
- § `Notification` represents the initial state of the automaton corresponding to the rule following the `HENCE` part.
- R represents the initial state of the automaton corresponding to the rule following the `LEST` part.

The purpose of x and q is to model a time-out, following the Time-out pattern described in [DHQ+08, Definition 12]. The idea is that once the guard condition (ie data breach occurs on or after ...) is satisfied, and the organization becomes aware, so that a becomeAware! transition occurs, we start a timer given by the fresh clock variable x and enter state q.

The clock invariant $x \leq 30$ enforces that when the automaton is in state q, we may only introduce a delay of up to 30 units of time before one of the 2 outgoing transitions must be taken. Taking 1 unit of time to be 1 day, this means that:

- If the organization assesses within 30 days, the `Assess` transition is taken so that the automaton is now in the § `Notification` state, which is the beginning of the corresponding rule.
- If 30 days passes and the organization doesn't assess, the transition to the `R` state becomes enabled and is thus taken (because the automaton cannot stay in state q anymore).

3.3.2 Automata to natural l4

This seems trickier.

Bibliography

[DHQ+08] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008.

[Lei16] Philip Leith. The rise and fall of the legal expert system. *International Review of Law, Computers & Technology*, 30(3):94–106, 2016.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1):134–152, 1997.