Effective Reasoning Systems for ASP Related Paradigms

by
Edward Wertz, B.S.

A Dissertation
in
Computer Science

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

Approved

Yuanlin Zhang
Chair of Committee

Michael Gelfond

Richard Watson

Mark Sheridan
Dean of the Graduate School

August, 2019
ACKNOWLEDGMENTS

I give sincere thanks and appreciation to Dr. Yuanlin Zhang for believing in me and serving as my committee chair. I give thanks and appreciation to Dr. Michael Gelfond for his many conversations and guidance. To the many graduate students that have helped me, thank you for your time and effort in helping me achieve milestones along the way. To my wife and kids, I thank you for your patience and sacrifice of time with me through this process.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 CALM</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Basic Action Theory (BAT)</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Sorted Signature</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Basic Action Theory (BAT)</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3 Semantics of BAT</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Syntax and Semantics of ALM System Description</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Syntax of ALM</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2 Semantics of ALM System Descriptions</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Design Of CALM</td>
<td>33</td>
</tr>
<tr>
<td>2.3.1 Background</td>
<td>33</td>
</tr>
<tr>
<td>2.3.2 Translation</td>
<td>35</td>
</tr>
<tr>
<td>2.3.3 Semantic Errors</td>
<td>42</td>
</tr>
<tr>
<td>2.3.4 Reasoning System</td>
<td>52</td>
</tr>
<tr>
<td>2.4 Implementation of CALM</td>
<td>56</td>
</tr>
<tr>
<td>2.4.1 Parsing System Descriptions and Tasks</td>
<td>56</td>
</tr>
<tr>
<td>2.4.2 Modeling a Basic Action Theory</td>
<td>57</td>
</tr>
<tr>
<td>2.4.3 Error Checking and Reporting</td>
<td>58</td>
</tr>
<tr>
<td>2.4.4 Translation to SPARC</td>
<td>62</td>
</tr>
<tr>
<td>2.4.5 System Usage</td>
<td>69</td>
</tr>
<tr>
<td>3 ICLP</td>
<td>70</td>
</tr>
<tr>
<td>3.1 Constraint Logic Programming (CLP)</td>
<td>72</td>
</tr>
<tr>
<td>3.1.1 CLP Program</td>
<td>72</td>
</tr>
</tbody>
</table>

iii
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2</td>
<td>Queries and Derivation Trees</td>
<td>74</td>
</tr>
<tr>
<td>3.2</td>
<td>The Incremental Query Problem</td>
<td>77</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Incremental Query</td>
<td>77</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The Incremental Query Problem And Solution</td>
<td>79</td>
</tr>
<tr>
<td>3.3</td>
<td>Records Of Computation</td>
<td>79</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Computation Trees and Paths Of Computation</td>
<td>79</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Record Of Computation</td>
<td>82</td>
</tr>
<tr>
<td>3.4</td>
<td>IQ Transition Diagram</td>
<td>83</td>
</tr>
<tr>
<td>3.4.1</td>
<td>IQ State Of Computation</td>
<td>83</td>
</tr>
<tr>
<td>3.4.2</td>
<td>IQ State Transitions</td>
<td>85</td>
</tr>
<tr>
<td>3.4.3</td>
<td>IQTD Path</td>
<td>94</td>
</tr>
<tr>
<td>4</td>
<td>CONCLUSION AND FUTURE WORK</td>
<td>97</td>
</tr>
<tr>
<td>4.1</td>
<td>CALM Conclusion</td>
<td>97</td>
</tr>
<tr>
<td>4.2</td>
<td>CALM Future Work</td>
<td>97</td>
</tr>
<tr>
<td>4.3</td>
<td>ICLP Conclusion</td>
<td>98</td>
</tr>
<tr>
<td>4.4</td>
<td>ICLP Future Work</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>APPENDICES</td>
<td>104</td>
</tr>
<tr>
<td>A</td>
<td>ALM Grammar in ANTLR4</td>
<td>104</td>
</tr>
<tr>
<td>B</td>
<td>CALM User and Developer Manual</td>
<td>121</td>
</tr>
<tr>
<td>C</td>
<td>IQTD Proofs</td>
<td>129</td>
</tr>
</tbody>
</table>
ABSTRACT

This dissertation focuses on improving the effectiveness of two Answer Set Programming (ASP) related paradigms. The first paradigm is ALM, a recent action language that allows the modular specification of ontologies and state transition diagrams to model actions and their effects in finite dynamic domains. Currently there is no publicly available compiler or reasoning system which accepts ALM System Descriptions as input. In this dissertation we describe and implement CALM, a compiler and reasoning system that translates ALM System Descriptions to SPARC, a variant of ASP. CALM enables the development of knowledge libraries and the investigation of best practices in modeling with ALM.

One future extension of ALM is to incorporate elements of action language H which enable reasoning about continuous change over time. In order to reason in continuous domains, programs in H are translated to AC(C), an extension of ASP to include constraint logic programming (CLP) reasoning and query answering in continuous domains. Our second area of focus is improving the effectiveness of the CLP algorithm used in AC(C) solvers. AC(C) solvers extend ASP solvers with an incrementally changing query to the CLP program derived from the AC(C) program. Current solvers restart CLP reasoning from scratch when the query changes, leading to redundant computation in the search for answers to the portions of the query that did not change. In this dissertation we formalize the incremental query problem and provide an incremental algorithm that reuses the solutions of previous queries in the search for answers to the modified query.
CHAPTER 1
INTRODUCTION

This dissertation focuses on improving the effectiveness of two Answer Set Programming (ASP) related paradigms. The first paradigm is ALM[24], a recent action language that allows the modular specification of ontologies and state transition diagrams to model actions and their effects in finite dynamic domains. Currently there is no publicly available compiler or reasoning system which accepts ALM System Descriptions as input. In this dissertation we describe and implement CALM, a compiler and reasoning system that translates ALM System Descriptions to SPARC[2, 1], a variant of ASP. CALM enables the development of knowledge libraries and the investigation of best practices in modeling with ALM.

One future extension of ALM is to incorporate elements of action language H[7, 6] which enable reasoning about continuous change over time. In order to reason in continuous domains, programs in H are translated to AC(C)[35, 36, 18], an extension of ASP to include constraint logic programming (CLP)[25, 26] reasoning and query answering in continuous domains. Our second area of focus is improving the effectiveness of the CLP algorithm used in AC(C) solvers. AC(C) solvers extend ASP solvers with an incrementally changing query to the CLP program derived from the AC(C) program. Current solvers restart CLP reasoning from scratch when the query changes, leading to redundant computation in the search for answers to the portions of the query that did not change. In this dissertation we formalize the incremental query problem and provide an incremental algorithm that reuses the solutions of previous queries in the search for answers to the modified query.

A dynamic domain can be viewed as a transition diagram whose nodes are possible states of the domain and whose arcs are actions in the domain. Action languages have been used to conveniently specify the state transition diagram, but are restricted to
small or medium sized domains [15]. \textit{ALM} is a recently developed modular action
language that addresses larger domains [24]. It supports the modeling by the concepts
of modules, module hierarchy and library.

In this dissertation we present \textbf{CALM} – a compiler for \textit{ALM}. It can trans-
late an \textit{ALM} system description \textit{P} into a \textit{SPARC} program which specifies the
same state transition diagram of \textit{P}. \textbf{CALM} also supports language for specifying
temporal projection and planning problems. \textbf{CALM} will translate a given system
description and the specification of a temporal projection or planning problem into a
\textit{SPARC} program whose answer sets contain solutions to these problems.

\textbf{CALM} is the first compiler to support the complete syntax of \textit{ALM} and per-
form semantic error checking prior to translation into an Answer Set Programming
(\textit{ASP}) program. Prior to \textbf{CALM}, there is a prototype translator for \textit{ALM} [24].
It only works with correct system descriptions (syntactically and semantically), It
does not implement all of the syntax of \textit{ALM}, but covers its significant core. An-
other modular action language is \textit{MAD}[29, 10, 30]. A \textit{MAD} compiler can trans-
late a \textit{MAD} program into a program in the language of the Causal Calculator
(\textit{CCALC})[19, 33] that can be used to carry out reasoning tasks.

One significant difference between \textit{ALM} and \textit{MAD} in modeling dynamic do-
ments is that the semantics of \textit{ALM} are based on the “law of inertia” [34] which
says that “things normally stay the same” while \textit{MAD} is founded on the causali-
ity principle which states that “everything true in the world must be caused”[14].
\textit{ALM} and \textit{MAD} also differ in their organization of modules, hierarchical ontologies,
and instantiation of objects[22, 10].

Other action languages of note include \textit{TAL−C} [21] and \textit{Modular BAT}[20].
While \textit{TAL−C} provides its semantics in translation to \textit{CCALC}, it does not present a
mechanism for specifying reusable modules of knowledge and formal organization of
a hierarchical ontology. \textit{Modular BAT} has similar goals to \textit{ALM}[22] but is based in
the situation calculus. ALM also contains a concept of basic action theory, but our semantics are provided through ASP.

Due to translation of ALM System Descriptions to ASP, CALM can only support finite domains for sorts. In order to support continuous domains and their corresponding constraint relations, ALM System Descriptions must be translated to a different language. The area of Constraint Answer Set Programming (CASP) extends ASP with constraint programming techniques which are capable of handling the real number domain and its constraint relations. The languages AC(C)[36], CLINGCON[40], and EZCSP[3, 4] have all addressed the task in various ways. Both EZCSP and CLINGCON are equivalent to AC−[28] a subset of AC(C).

The ACSolver for AC(C) incorporates both ASP and CLP solvers to handle the regular and constraint portions of the program. The original ACSolver[35] executes the CLP query from scratch when the query is extended or backtracking occurs. The LUNA solver[37] improved the performance of the ACSolver by reusing the answer to the previous query when the query to the CLP program was extended. If backtracking occurred and the query was reduced, the query would be re-executed against the CLP program. This re-execution of the query during backtracking performs redundant computation. In order to eliminate this redundant computation, the CLP solver must be modified to become an incremental solver which preserves solutions to incrementally constructed queries for reuse when the query is decremented during backtracking. For a CLP solver, both ACSolver and LUNA used CLAM(\mathcal{R})[27], an extension of the WAM[44] to include arithmetic constraints over real numbers. Our eventual goal is to provide a modified instruction set and implementation to the CLAM(\mathcal{R}) solver which saves results of incrementally constructed queries for later reuse. As a significant step towards that goal we have developed a state transition diagram called IQTD which models the state of an incremental solver as it searches the sld-derivation tree for the answers to an incremental query of
a logic program. As our goal was to extend the capability of the CLAM(\(\mathcal{R}\)) solver, we did not investigate the use of tabled logic programs\[5\] and slg-resolution.

The ACSolver’s query to the CLP solver behaves as a stack. The query is extended by appending a new query increment as a result of unit propagation in the ACSolver and the most recently appended query is removed when backtracking removes the context which forced the query to be extended. Our criteria for reviewing the existing approaches to incremental CLP solvers included the requirements that it can be implemented as a modification to CLAM(\(\mathcal{R}\)), support a stack like behavior for query modification, and preserve all solutions discovered for each prefix of the active query stack to eliminate redundant computation during backtracking.

In our literature review we investigated several approaches to incremental constraint logic programming which we could not apply given our criteria. The Reactive CLP scheme\[11, 12\] did account for the query manipulating operations, but the solution performs transformations of the sld-derivation tree which would require significant redesign of the CLAM(\(\mathcal{R}\)) solver and its proofs of correctness. Many incremental query efforts provide their solutions as meta-interpreters implemented in Prolog\[42, 38\] and their translation to a machine instruction set would not be straightforward.

Our solution IQTD extends the work of Peter Stuckey’s Expanding query power in constraint logic programming languages\[31\] and Pascal Van Hentenryck’s Incremental search in constraint logic programming\[43\]. Stuckey’s work saves every solution encountered for a query and executes a query extension in the context of each saved solution to derive solutions for the extended query. Stuckey did not address removal of queries except to indicate re-execution of the remaining query would be needed. Van Hentenryck’s work re-executes portions of a path of computation when constraints are added or removed from the query, but his work does not address addition and removal of atoms. Our solution combines both approaches. At every level of the
incremental query we save all solutions discovered and their paths of computation for
future reuse after removal of queries.
CHAPTER 2
CALM

2.1 Basic Action Theory (BAT)

In this chapter we recall the concept of a Basic Action Theory (BAT) [24] by first introducing sorted signature and then the syntax and semantics of BAT.

2.1.1 Sorted Signature

The sorted signature of a BAT is $\Sigma = \langle C, O, H, F \rangle$ where $C$, $O$, and $F$ are sets of strings over some fixed alphabet, and

1. $C = C_{sp} \sqcup C_{pd} \sqcup C_{ud}$ where $C_{sp} = \{\text{nodes, obj constants, universe}\}$ elements of $C_{sp}$ are called special sorts; $C_{pd} = \{\text{boolean, integer, [m..n]}\}$, where $m, n$ are natural numbers and $m < n$, and elements of $C_{pd}$ are called pre-defined sorts, and $C_{ud}$ is a set and elements of $C_{ud}$ are called user defined sorts. Elements of $C$ are called sort names in general.

2. $O = O_{pd} \sqcup O_{ud}$ where $O_{pd} = \{\text{true, false, 0, 1, -1, 2, ...}\}$, and elements of $O_{pd}$ are called predefined object constants, and $O_{ud}$ is a set and its elements are called user defined object constants. Elements of $O$ are called object constants.

3. $H$ is a directed acyclic graph $(V, E)$, where $V \subseteq (\{\text{universe}\} \cup C_{pd} \cup C_{ud}) \cup O$. $H$ is called a sort hierarchy. Sometimes $(C, O, H)$ is called an ontology. For simplicity, we assume the graph has exactly one sink node.

(a) For any $(c_1, c_2) \in E$, where $c_1, c_2 \notin O$, $c_2$ is called parent of $c_1$. For any sort names $c_1$ and $c_2$, $c_1$ is a subsort of $c_2$ if $c_1$ is a descendant of $c_2$ in $H$.

(b) For any $(o, c)$ where $c \notin O$ an $o \in O$, $o$ is called of sort $c$. For any parent $c_2$ of $c_1$, if $o$ is of sort $c_1$, then $o$ is of sort $c_2$. 

6
4. $F = F_{pd} \sqcup F_{sp} \sqcup F_{ud}$ where

- $F_{pd} = \{+, -, \times, /, \leq, \geq, <, >, \ldots\}$, and elements of $F_{pd}$ are called the *pre-defined* functions.

- $F_{sp} = F_{spH} \sqcup F_{spD}$ where
  
  $F_{spH} = \{\text{link} : \text{nodes} \times \text{nodes} \to \text{boolean};$
  
  $\text{is}_a : \text{universe} \times \text{nodes} \to \text{booleans};$
  
  $\text{instance} : \text{universe} \times \text{nodes} \to \text{booleans};$
  
  $\text{subsort} : \text{nodes} \times \text{nodes} \to \text{boolean};$
  
  $\text{has}_\text{child}, \text{has}_\text{parent}, \text{sink}, \text{source} : \text{nodes} \to \text{boolean}\}$, and

  $F_{spD} = \{(\text{dom}_f : c_1, \ldots, c_n \to \text{boolean}) \mid (f : c_1, \ldots, c_n \rightarrow c) \in F_{ud}\}.$

Elements of $F_{spH}$ are called *sort hierarchy functions*. Elements of $F_{spD}$ are called *domain functions*. Elements of $F_{sp}$ are called *special functions*.

- $F_{ud}$ is a set of functions, and its elements are called *user defined* functions.

- We assume a function $f \in F$ of the form $f : c_1, \ldots, c_n \to c$ where $f$ is a string, $c$ and $c_i$ ($i \in 1..n$) are sort names. $f$ is called the *name* of the function, $n$ is called the *arity* of $f$, $c_1, \ldots, c_n$ are called the *sorts of parameters* or *domain sorts* of $f$, and $c$ is called the *range* or *range sort* of $f$. A domain sort or range sort of $f$ is called a *signature sort* of $f$.

A variable and an object constant is a *term*. If $f : c_1 \times c_1, \ldots, c_n \rightarrow c$ is an element of $F$, and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots t_n)$ is a *term*.

Expressions of the form $t_1 = t_2$ and $t_1 \neq t_2$, where $t_1$ and $t_2$ are terms, are called *literals*. The former is also called *atoms*. We use standard shorthand and write $t$ and $\neg t$ instead of $t = \text{true}$ and $t = \text{false}$.

Terms and literals not containing variables are called *ground*.  

7
2.1.2 Basic Action Theory (BAT)

We first introduce action signatures. An action signature is a sorted signature

\[ \Sigma = \langle C, O, H, F \rangle \]

where \( C \) includes the string actions, there is a node labeled by actions and an arc labeled by \( \langle \text{actions, universe} \rangle \) in \( H \). In an action signature, its user-defined and special function symbols are divided into three disjoint categories: attributes, statics, and fluents. Both statics and fluents are further divided into basic and defined. The latter are total boolean functions that can be defined in terms of the former.

We next introduce the definitions of axiom statements.

- A dynamic causal law is an expression of the form

\[ \text{occurs}(a) \ \text{causes} \ f(\bar{x}) = o \ \text{if} \ \text{instance}(a, c), \text{cond} \]

(2.1)

where \( \text{occurs}, \ \text{causes} \) and \( \text{if} \) are keywords, \( a \) and \( o \) are variables or object constants and \( \bar{x} \) is a sequence of terms, \( f \) is a basic fluent, \( c \) is the sort actions or a subsort of it, and \( \text{cond} \) is a collection of literals.

The law says that an occurrence of an action \( a \) of the sort \( c \) in a state satisfying property \( \text{cond} \) causes the value of \( f(\bar{x}) \) to become \( o \) in any resulting state.

- A state constraint is an expression of the form

\[ f(\bar{x}) = o \ \text{if} \ \text{cond} \]

(2.2)

where
$o$ is a variable or an object constant, $f$ is any function except a defined function, and $cond$ is a collection of literals.

The law says that the value of $f(\bar{x})$ in any state satisfying condition $cond$ must be $o$.

Additionally, $f(\bar{x}) = o$ can also be replaced by the object constant $false$, in which case the law says that there is no state satisfying condition $cond$.

• The definition of a defined function $p$ is an expression of the form

$$p(\bar{t}_1) \text{ if } cond_1$$
$$\ldots$$
$$p(\bar{t}_k) \text{ if } cond_k$$

(2.3)

where $\bar{t}_1, \ldots, \bar{t}_k$ are sequences of terms, and $cond_1, \ldots, cond_k$ are collections of literals. Moreover, if $p$ is a static then $cond_1, \ldots, cond_k$ can not contain fluent literals. Statements of the definition will be often referred to as its clauses.

The statement says that, for every $\bar{Y}$, $p(\bar{Y})$ is true in a state $\sigma$ iff there is $1 \leq m \leq k$ such that statements $cond_m$ and $\bar{t}_m = \bar{Y}$ are true in $\sigma$.

• An executability condition for actions is an expression of the form

$$\text{impossible occurs}(a) \text{ if } instance(a,c), cond$$

(2.4)

where impossible is a keyword, $a$ is a variable or an object constant, $c$ is the sort actions or a subsort of it, and $cond$ is a collection of literals and expressions of the form $occurs(t)$ or $\neg occurs(t)$ where $t$ is a variable or an object constant of the sort actions.
This law says that an occurrence of an action of the sort is impossible when condition holds.

Dynamic causal laws and constraints will be sometimes referred to as causal laws. We use the term head to refer to the literal immediately before if in (2.1) and (2.2), and to any of the \( p(t_i), 1 \leq i \leq k \), in (2.3). We call body the expression to the right of the keyword if in statements (2.1), (2.2), (2.4), or in any of the statements of (2.3). Statements not containing variables will be referred to as ground.

An axiom statement or axiom is dynamic causal law, a state constraint, a definition of a defined function, or an executability condition.

A basic action theory (BAT) is a pair \((\Sigma, A)\) where \(\Sigma\) is an action signature, and \(A\) is a set of axioms over this signature satisfying the following:

- If \( f \) is a basic fluent then

  - \( A \) contains a state constraint:

    \[
    \text{dom}_f(X_0, \ldots, X_n) \text{ if } f(X_0, \ldots, X_n) = Y
    \]  
    \[\text{(2.5)}\]

  - No dynamic causal law of \( A \) contains an atom formed by \( \text{dom}_f \) in the head.

- If \( f \) is a defined fluent, a static, or an attribute then \( A \) contains the definition:

  \[
  \text{dom}_f(X_0, \ldots, X_n) \text{ if } f(X_0, \ldots, X_n) = Y
  \]  
  \[\text{(2.6)}\]

- \( A \) contains definitions of special statics of the hierarchy given in terms of func-
2.1.3 Semantics of BAT

We introduce the semantics of BAT by first introducing the interpretation of a sorted signature and then the model of a BAT.

2.1.3.1 Interpretation of Sorted Signatures

Definition 1. (Interpretation)

An Interpretation $I$ of a sorted signature $\Sigma = \langle C, O, H, F \rangle$ consists of

- A non-empty set $|I|$ of strings called the universe of $I$.

- An assignment that maps

  - every user-defined sort $c$ of $H$ into a subset $I(c)$ of $|I|$ such that
    
    * if $\langle c_1, c_2 \rangle \in H$ then $I(c_1) \subseteq I(c_2)$ and
    
    * $I(c) = \bigcup \{I(c_i) : c_i \text{ is a child of } c \text{ in } H\}$

  - every object constant $o$ of $\Sigma$ into an element of $|I|$ such that if $(o, c) \in H$ then $I(o) \in I(c)$;
– every user-defined function symbol \( f : c_1 \times \cdots \times c_n \to c_0 \) of \( \Sigma \) into a (possibly partial) function \( I(f) : I(c_1) \times \cdots \times I(c_n) \to I(c_0) \);

– The special function \( \text{is}_a \) into a function \( I(\text{is}_a) : |I| \times \text{nodes} \to \text{booleans} \) such that for every \( x \in |I| \) and every sort \( c \in \text{nodes} \), \( I(\text{is}_a)(x, c) \) is \textit{true} iff \( c \) is a source node of \( H \) and \( x \in I(C) \);

– the special function \( \text{link} \) into function \( I(\text{link}) : \text{nodes} \times \text{nodes} \to \text{booleans} \) such that for every two sort nodes \( c_1, c_2 \), \( I(\text{link})(c_1, c_2) \) is \textit{true} iff \( \langle c_1, c_2 \rangle \in H \);

– the special function \( \text{dom}_f \) for user-defined function \( f : c_1 \times \cdots \times c_n \to c_0 \) into function \( I(\text{dom}_f) \) such that for every \( \bar{x} \in I(c_1) \times \cdots \times I(c_n) \), \( I(\text{dom}_f)(\bar{x}) \) is \textit{true} iff \( \bar{x} \) belongs to the domain of \( I(f) \).

\begin{itemize}
  \item On pre-defined symbols, \( I \) is identified with the symbol’s standard interpretations.
\end{itemize}

\[ \square \]

Let signature \( \Sigma \) and an interpretation \( I \) be given. \( I \) can be partitioned into two parts: the \textit{fluent part} consisting of the universe of \( I \) and the restriction of \( I \) on the sets of fluents, and the \textit{static part} consisting of the same universe and the restriction of \( I \) on the remaining elements of the signature. The static part of \( I \) is referred to as the \textit{static interpretation of} \( \Sigma \).

Let signature \( \Sigma \) and a collection of strings \( U \) in some fixed alphabet be given. \( \Sigma_U \) denotes the signature obtained from \( \Sigma \) by expanding its set of object constants by elements of \( U \), which we assume are of sort \textit{universe}.

### 2.1.3.2 Models of BAT

Given an interpretation \( I \) of an action signature \( \Sigma \), we define \textit{fluent part} of it consisting of the universe of \( I \) and the restriction of \( I \) on the sets of fluents. We define \textit{static part}
of \( I \) consisting of the same universe and the restriction of \( I \) to the remaining elements of the signature. Sometimes we will refer to the latter as a static interpretation of \( \Sigma \).

Given an action signature \( \Sigma \) and a collection \( U \) of strings in some fixed alphabet, we denote by \( \Sigma_U \) the signature obtained from \( \Sigma \) by expanding its set of object constants by elements of \( U \), which we assume to be of sort universe.

**Definition 2.** (Pre-Model)

Let \( T \) be a basic action theory with signature \( \Sigma \) and \( U \) be a collection of strings over some fixed alphabet. A static interpretation \( M \) of \( \Sigma_U \) is called a pre-model of \( T \) (with the universe \( U \)) if \( M(\text{universe}) = U \) and for every object constant \( o \) of \( \Sigma_U \) that is not an object constant of \( \Sigma \), \( M(o) = o \).

A pre-model \( M \) for basic action theory \( T \) defines a model, a state transition diagram, \( T_M \). We define \( T_M \) through its states and transitions.

**Definition 3.** (Program \( S_M \))

Let \( M \) be a pre-model of basic action theory \( T \). By \( S_M \) we denote the logic program that consists of:

- rules obtained from the state constraints and definitions of \( T \) by replacing variables with the properly typed object constants of \( \Sigma_M \), replacing object constants with their corresponding interpretations in \( M \), removing the constant \( false \) from the head of state constraints, and replacing the keyword \( if \) with \( \leftarrow \),

- The Closed World Assumption:

\[
-d(t_1, \ldots, t_n) \leftarrow \text{not } d(t_1, \ldots, t_n).
\]

for every defined function \( d : c_1 \times \cdots \times c_n \rightarrow \text{booleans} \) and \( t_i \in M(c_i), 1 \leq i \leq n \).
Definition 4. (Program $S_I$)
For every interpretation $I$ of $\Sigma$ with static part $M$, by $S_I$ we denote the logic program obtained by adding to $S_M$ the set of atoms obtained from $I$ by removing the defined atoms.

Definition 5. (State)
Let $M$ be a pre-model of a basic action theory $T$. An interpretation $\sigma$ with static part $M$ is a state of the transition diagram $T_M$ defined by $M$ if $\sigma$ is the only answer set of the logic program $S_{\sigma}$.

Definition 6. (Program $P_M$)
Program $P_M$ is obtained from a basic action theory $T$ and pre-model $M$ by

1. replacing variables by properly typed object constants of $\Sigma_M$;
2. replacing object constants by their corresponding interpretation in $M$;
3. removing the object constant $false$ from the head of state constraints;
4. replacing every occurrence of a fluent term $f(\bar{t})$ in the head of a dynamic causal law by $f(\bar{t}, I + 1)$;
5. replacing every other occurrence of a fluent term $f(\bar{t})$ by $f(\bar{t}, I)$;
6. removing “occurs($a$) causes” from every dynamic causal law and adding $\neg$occurs($a$) to the body.
7. replacing “impossible occurs($a$)” in every executability condition by $\neg$occurs($a$);
8. replacing $occurs(a)$ by $occurs(a, I)$ and $\neg occurs(a)$ by $\neg occurs(a, I)$;

9. replacing the keyword if by ←;

10. adding the Closed World Assumption:

$$\neg d(t_1, \ldots, t_n, I) \leftarrow \text{not} d(t_1, \ldots, t_n, I)$$

for every defined fluent $d : c_1, \times \cdots \times c_n \rightarrow booleans$ and $t_i \in M(c_i)$, $1 \leq i \leq n$;

11. adding the rule:

$$\neg f(t_1, \ldots, t_n) \leftarrow \text{not} f(t_1, \ldots, t_n)$$

for every defined static of the form $f : c_1 \times \cdots \times c_n \rightarrow booleans$ and $t_i \in M(c_i)$, $1 \leq i \leq n$;

12. adding the Inertia Axiom:

$$\text{dom}_f(t_1, \ldots, t_n, I + 1) \leftarrow \text{dom}_f(t_1, \ldots, t_n, I), \text{not} \neg \text{dom}_f(t_1, \ldots, t_n, I + 1)$$

$$\neg \text{dom}_f(t_1, \ldots, t_n, I + 1) \leftarrow \neg \text{dom}_f(t_1, \ldots, t_n, I), \text{not} \text{dom}_f(t_1, \ldots, t_n, I + 1)$$

for every basic fluent $\text{dom}_f : c_1 \times \cdots \times c_n \rightarrow booleans$, and $t_i \in M(C_i)$, $1 \leq i \leq n$;

13. adding the Inertia Axiom:

$$f(t_1, \ldots, t_n, I + 1) = t \leftarrow \text{dom}_f(t_1, \ldots, t_n, I + 1), f(t_1, \ldots, t_n, I) = t,$$

$$\text{not} f(t_1, \ldots, t_n, I + 1) \neq t$$

for every basic fluent $f : c_1 \times \cdots \times \rightarrow c_0$ not formed by $\text{dom}$, and $t_i \in M(c_i)$, $1 \leq i \leq n$, and $t \in M(c_0)$.
**Definition 7.** (Program $P(M, \sigma_0, a)$)

Let $\sigma_0$ be a state of the transition diagram defined by a pre-model $M$, and let $a \subseteq M(\text{actions})$. By $P(M, \sigma_0, a)$ we denote the logic program formed by adding to $P_M$ the set of atoms obtained from $\sigma_0$ by replacing every fluent atom $f(t_1, \ldots, t_n) = t$ by $f(t_1, \ldots, t_n, 0) = t$ and adding the set of atoms $\{\text{occurs}(x, 0) : x \in a\}$.

□

**Definition 8.** (Transition)

Let $\sigma_0$ and $\sigma_1$ be states of the transition diagram defined by a pre-model $M$ and let $a \subseteq M(\text{actions})$. The triple $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition of the transition diagram defined by a pre-model $M$ of a basic action theory $T$ if program $P(M, \sigma_0, a)$ has an answer set $A$ such that $f(t_1, \ldots, t_n) = t \in \sigma_1$, iff

- $f$ is an attribute or static and $f(t_1, \ldots, t_n) = t \in A$, or
- $f$ is a fluent and $f(t_1, \ldots, t_n, 1) = t \in A$.

□

**Definition 9.** (Model)

A transition diagram $T_M$ defined by pre-model $M$ of a basic action theory $T$ is called a model of $T$ if it has a non-empty collection of states.

□

2.2 Syntax and Semantics of $\mathcal{ALM}$ System Description

The $\mathcal{ALM}$ paper presents $\mathcal{ALM}$ through examples. In this chapter we will make a more explicit treatment of the syntax and semantics of $\mathcal{ALM}$. The complete ANTLR4 grammar is contained in Appendix A.
2.2.1 Syntax of $\mathcal{ALM}$

2.2.1.1 Syntax of $\mathcal{ALM}$ Theories

A sort declaration is of the form:

$$id_1, \ldots, id_n :: sort_1, \ldots, sort_m$$

$$attr_1 : c_{1,1} \times \cdots \times c_{1,k_1} \to c_{1,0}$$

$$\cdots$$

$$attr_l : c_{l,1} \times \cdots \times c_{l,k_l} \to c_{l,0}$$

where for $i \in [1..n]$, $j \in [1..m]$, $id_i$ and $sort_j$ are identifiers and called sort names and $id_i$ is called a declared sort and a declared subsort of $sort_j$. All $attr_1, \ldots, attr_l$ are optional attribute function declarations. The syntax for attribute function declaration is a shorthand: for $i \in [1..m], j \in [1..l]$, the signature of the attribute function has the form $attr_j : sort_i \times c_{j,1} \times \cdots \times c_{j,k_j} \to c_{j,0}$. If $sort_j$ is the only domain argument to the signature, the attribute declarations have the form $attr_j : c_{j,0}$.

A sort declarations section is of the form

$$\text{sort declarations } sd_1 \ldots sd_n$$

where sort declarations are reserved identifiers, $sd_i$ is a sort declaration.

We next define constant declaration which introduces a collection of named object constants and indicates to which sorts they belong.

A constant declaration is of the form

$$id_1, \ldots, id_n :: sort_1, \ldots, sort_m$$

where for $i \in [1..n]$, $j \in [1..m]$, $id_i$ is called a declared constant and $sort_j$ is called a declared sort of constant $id_i$. A constant declarations section is of the form

$$\text{constant declarations } cd_1 \ldots cd_n$$

where constant declarations are reserved identifiers, $cd_i$ is a constant declarations.
A function declaration is of the form

\[ \text{[total]} \ f : \text{sort}_1, \ldots, \text{sort}_n \rightarrow \text{sort} \]

where \( f \) is an identifier and called function name and for \( i \in [1..n] \), \text{sort}_i \) is a sort name. Each function declaration is also called a user-defined function signature. A functions declaration section is of the form

function declarations

statics

basic \( \text{bf}_1 \ldots \text{bf}_{n_1} \)
defined \( \text{bd}_1 \ldots \text{bd}_{n_2} \)

fluents

basic \( \text{ff}_1 \ldots \text{ff}_{n_3} \)
defined \( \text{fd}_1 \ldots \text{fd}_{n_4} \)

where function declarations, statics, basic and defined are reserved words, \( n_1, \ldots, n_4 \) are numbers, and every \( \text{bf}_i, \text{bd}_i, \text{ff}_i \) and \( \text{fd}_i \) is a function declaration.

An axioms section is of the form

axioms \( a_1 \ldots a_n \)

where axioms is a reserved identifier, and \( a_i \) is an axiom as defined in BAT.\(^1\)

A module dependencies section is of the form

depends on \( M_1, \ldots, M_n \)

where \( M_i \) is a module name (which will be defined later).

A module is of the form

module \( \text{mname} \ mdep \ sdec \ cdec \ fdec \ axioms \)

\(^1\)The notion of literals is used in the definition of axioms in BAT. This notion can be redefined using the terminologies here, but we do not repeat the definitions.
where module is a reserved identifier, mname is an identifier and called a module name, mdep is a module dependencies section, sdec is a sort declarations section, cdec is a constant declarations section, fdec is a function declarations section, and axioms is an axioms section.

A theory is of the form

\[ \text{theory } tname \ m_1 \ldots \ m_n \]

where theory is a reserved identifier, tname is an identifier and called a theory name, and \( m_i \) is a module.

2.2.1.2 Syntax of ALM Structures

A structure consists of constant definition, instance definition, instance schema definition, and statics definition. Those concepts will be defined below in order.

A constant definition is of the form

\[ id = \text{groundterm} \]

An instance definition is of the form

\[ \text{groundterm in sort.} \]

An instance schema definition is of the form
\[ id(V_1, \ldots, V_n) \text{ in } c \text{ where } l_1, \ldots, l_m \]
\[ f_1(\vec{t}_1) = V_1 \]
\[ \ldots \]
\[ f_n(\vec{t}_n) = V_n \]

where \( f_1, \ldots, f_n \) are attribute functions declared for \( c \) and its super sorts, \( V_1, \ldots, V_n \) are variables, the sort of \( V_i \) is the range sort of \( f_i \), \( l_1, \ldots, l_m \) are literals, and \( \vec{t}_1, \ldots, \vec{t}_n \) are vectors of terms. If a vector \( \vec{t}_i \) is empty, the containing parenthesis ( ) are removed.

A \textit{statics definition} is of the form

\[ l_0 \text{ if } l_1, \ldots, l_n. \]

where \( l_0 \) is a static function literal and \( l_1, \ldots, l_n \) are static literals.

A \textit{structure} is of the form

\textit{structure} \ sname \ consdef \ insdef \ statdef

where \textit{structure} is a reserved identifier, \textit{sname} is an identifier and called a \textit{structure name}, \textit{consdef} is a sequence of constant definitions, \textit{insdef} is a sequence of instance definitions and instance schema definitions, and \textit{statdef} is a sequence of statics definitions.

2.2.1.3 ALM System Descriptions

An \textit{ALM system description} is of the form

\textit{system description} \ name \ theory \ structure
where **system description** are reserved identifiers, **name** is an identifier and called a **system description name**, **theory** is a theory and **structure** is a structure.

### 2.2.1.4 Well Defined System Descriptions

In this dissertation, we consider only a special class of system descriptions. First, we define the notion of sort dependency.

**Definition 10.** The Sort Dependency Graph of a System Description

Given a system description $P$ with core BAT $T = (\Sigma = \langle C, O, H = (V, E), F \rangle, A)$, the **sort dependency graph** is the minimal directed graph $(V', E')$ such that $V' \subset V$ and $E'$ is defined as follows:

- $\langle c_2, c_1 \rangle \in E'$ where $\langle c_1, c_2 \rangle \in E$ and $c_1 \in C$.
- For any instance schema definition of $P$ with the following form:

  $$
  \text{id}(V_1, \ldots, V_n) \text{ in } c_1 \text{ (where } l_1, \ldots, l_m\text{)}?
  \begin{align*}
  f_1(\bar{t}_1) & = V_1 \\
  \vdots \\
  f_n(\bar{t}_n) & = V_n
  \end{align*}
  $$

  $\langle c_1, c_2 \rangle \in E'$ where $c_2$, $c_2 \neq c_1$, is user defined signature sort of some attribute function $f_i$ ($i \in [1..n]$), or a user defined signature sort of a function occurring in $l_1, \ldots, l_m$, or occurs in any sort hierarchy functions that occurs in $l_1, \ldots, l_m$.

- For any static function definition $r$ in the structure of $P$ such that $c_1$ is a signature sort of a function occurring in the head of $r$ and $c_2$ is a signature sort of a function occurring in the body of $r$

A sort $c_2$ **depends on** $c_1$ if there is a path from $c_2$ to $c_1$ in $(V', E')$. 

\[\square\]
**Definition 11.** Well Defined System Descriptions

A system description \( P \) is *well defined* if its sort dependency graph is acyclic.

\[ \square \]

2.2.2 Semantics of ALM System Descriptions

In the following treatment of the semantics of ALM system descriptions, we assume a single module theory is present. A multi-module theory can be flattened into a single module [24].

An ALM system description \( P \) defines a BAT and its premodel(s).

2.2.2.1 Core BAT Defined by a System Description

**Definition 12.** The Core BAT Defined by the Theory of a System Description

Given a System Description \( P \), the core basic action theory \( T = (\Sigma = \langle C, O, H, F \rangle, A) \) defined by \( P \)'s theory is as follows:

- **\( C = C_{sp} \cup C_{pd} \cup C_{ud} \)** where \( C_{sp} \) and \( C_{pd} \) are the set of special sort names and predefined sort names as previously defined for BAT, and \( C_{ud} = \{ s \mid \text{where } s \text{ is a declared sort in } P \} \),

- **\( O = O_{pd} \cup O_{ud} \)** where \( O_{pd} \) is the set of predefined constants as previously defined for BAT and \( O_{ud} = \{ o \mid \text{where } o \text{ is a declared constant in } P \} \),

- **\( H = (V, E) \)** is the directed acyclic graph such that the set of nodes \( V \) = \{universe, actions\} \( \cup \) \( C_{pd} \cup C_{ud} \cup O \) and the set of edges \( E = E_{sp} \cup E_{pd} \cup E_{ud} \) where \( E_{sp} \) and \( E_{pd} \) are as defined previously for BAT and \( E_{ud} = \{ \langle v_1, v_2 \rangle \mid \text{either } v_1 \in C_{ud}, v_2 \in C_{ud} \cup \{ \text{actions, universe} \} \text{ and } v_1 \text{ is a declared subsort of } v_2 \text{ in } P, \text{ or, } v_1 \in O_{ud}, v_2 \in C_{ud} \text{ and } v_2 \text{ is a declared sort for constant } v_1 \text{ in } P \} \)

- **\( F = F_{sp} \cup F_{pd} \cup F_{ud} \)** where \( F_{pd} \) is as previously defined for BAT, \( F_{ud} = \{ f : c_1 \times \cdots \times c_n \rightarrow c \mid f : c_1 \times \cdots \times c_n \rightarrow c \text{ is a user-defined function signature in } \} \)
$P}$, and the previous definition of $F_{sp}$ for BAT is extended by the set \( \{ \text{dom}_{f}: c_1 \times \cdots \times c_n \rightarrow \text{booleans} \mid f : c_1 \times \cdots \times c_n \rightarrow c \in F_{ud} \} \), and

- $A = A_{sp} \cup A_{ud}$ where $A_{sp}$ are axioms from equations (2.5) to (2.7), and $A_{ud} = \{ a \mid a$ is a user-defined axiom in $P$ \}.

\[\square\]

**Proposition 1.** The core BAT defined by the theory of a system description is a BAT.

Proof. By comparing the definition of BAT and Definition 12, we can verify that this proposition holds.

\[\square\]

2.2.2.2 Static Interpretations Defined by ALM System Descriptions

An ALM system description is designed to specify an interpretation. We focus here on the static portion of the interpretation. The following diagram illustrates the key components leading to a static interpretation: the signature defined by a system description, the extended signature defined by such a description and the establishment of the \texttt{is_a} relation.
Due to the possible interactions among the instance schema definition, the state constraints on static functions and the statics definitions, we will use an ASP program to define the static interpretation specified by a system description.

Given an ALM system description $P$, let $T = (\Sigma = \langle C, O, H, F \rangle, A)$ be the core BAT defined by the theory of $P$. We need the following predicates:

- $\text{universe}(X)$ meaning $X$ is an element of the universe $U$ to be specified,
- $\text{constantInTheory}(X)$ meaning $X$ is a constant declared in the theory of $P$,
- $\text{extendedConstants}(X)$ meaning $X$ is an extended constant, i.e., a constant from constant declaration, constant definition or instance schema definition,
- $I_{\text{extendedConstants}}(X, Value)$ meaning extended constant $X$ is mapped to $Value$ of universe $U$,
- $I_{\text{sorts}}(o, c)$ meaning the interpretation of the sorts, i.e. $o$ is an element of sort $c$, 

* Dependency exists between instance schema, state constraints and statics def. ** All arrowed lines mean obtained from.
• \textit{link\_constantInTheory}(o, c) meaning that \((o, c) \in H\),

• \textit{link\_constantInStructure}(o, c) meaning that \((o, c)\) occurs in an instance schema definition or a constant definition,

• \textit{link\_extendedConstant}(o, c) meaning that an object sort edge \((o, c)\) occurs either in \(H\) or in structure,

• \textit{consInTheoryDef}(id, groundterm) meaning that the \(id = \text{groundterm}\) occurs in the structure,

• \textit{link}(c_1, c_2) meaning that \((c_1, c_2) \in H\).

2.2.2.3 ASP Program \(\Pi_1\) for Universe and Extended Signature

We define \(\Pi_1\) as follows:

• Represent \(H\) from the core BAT of \(P\).

  \[
  \text{\textit{link\_constantInTheory}}(X, C) :\text{-}\ \text{\textit{link\_constantInTheory}}(X, C).
  \]

  \[
  \text{\textit{link}}(c_1, c_2).
  \]

  \[
  \text{\textit{consInTheoryDef}}(id, \text{groundterm})
  \]

  % syntacticConstants from H

  \[
  \text{\textit{constantInTheory}}(X) :\text{-}\ \text{\textit{link\_constantInTheory}}(X, C).
  \]

  % syntacticConstants from H

  \[
  \text{\textit{constantInTheory}}(X) :\text{-}\ \text{\textit{link\_constantInTheory}}(X, C).
  \]

• Define syntactic constants from constant definition of the structure of \(P\).

  % the constant definition \(id = \text{groundterm}\) in structure is represented as

  \[
  \text{\textit{consInTheoryDef}}(id, \text{groundterm}).
  \]
%% We introduce \textit{consInTheoryDefined}(X) meaning syntactic constant \(X\) is defined in structure

\[ \text{consInTheoryDefined}(X) : \text{-} \text{consInTheoryDef}(X, \text{Term}). \]

- Define/Represent other user-defined elements of universe. They may be from constant definition or instance (schema) definition in the structure, or from constants not defined in the structure.

%% 3-1 for every constant definition \(id = \text{groundterm}\) such that the declared sort of \(id\) is \(c\),

\[ \text{link}_\text{constantInStructure} (\text{groundterm}, c). \]

%% 3-2 for every instance definition \(t \in c\)

\[ \text{link}_\text{constantInStructure} (t, c). \]

%% 3-3 for every instance schema \(id(V1, \ldots, Vn) \in c\) where \(l_1, \ldots, l_m\) (with) \(f_i(t_i) = V_i, 1 \leq i \leq n\).

\[ \text{link}_\text{constantInStructure} (id(V1, \ldots, Vn), c) : \]
\[ l_1, \ldots, l_m, \]
\[ \text{instance}(t_1, c_{i1}), \ldots, \text{instance}(t_n, c_{in}), \]
\[ \text{instance}(V1, c_{21}), \ldots, \text{instance}(Vn, c_{2n}). \]

where \(c_{ii}\) are the domain sorts of \(f_i\) and \(c_{2i}\) is the range sort of \(f_i\).

%% This schema also defines the attribute functions. For \(i \in 1..n\),

\[ f_i(id(V1, \ldots, Vn), Vi) : \text{-} \text{link}_\text{constantInStructure} (id(V1, \ldots, Vn), c). \]

% universe is all the instances in instance (schema) definitions in structure, i.e.,

\[ \text{link}_\text{explicitUniverseConstants}, \text{and the syntactic constants not defined in the structure.} \]

\[ \text{universe}(X) : \text{-} \text{link}_\text{constantInStructure} (X, C). \]
universe(X) :- constantInTheory(X), not consInTheoryDef(X).

- Extended constants of the extended signature with universe

  % extendedConstants are the union of syntactic constants and explicit element
  % in universe

  extendedConstants(X) :- constantInTheory(X).
  extendedConstants(X) :- link_constantInStructure(X,C).

2.2.2.4 ASP Program $\Pi_2$ for Interpretation of Extended Constants, is.a, and Sorts

- Map from extended constants to universe

  % the map as defined in the constant definitions in structure

  $I_{\text{extendedConstants}}(Id, \text{Groundterm})$ :-
  \begin{align*}
  & \text{constantInTheory}(Id), \\
  & \text{consInTheoryDef}(Id, \text{Groundterm}).
  \end{align*}

  % for element in the universe, it is mapped to itself

  $I_{\text{extendedConstants}}(Id, Id) :-$ universe(Id).

- Object-sort links in the hierarchy on extended constants. Note that there is NO
object-sort link between a defined syntactic constants and any sort.

  % build link between universe constants and sorts

  % 2-1 link between explicit universe constants and their sorts

  link_extendedConstant(O,C) :- link_constantInStructure(O,C).

  % 2-2 link for H where syntactic constants are not defined in structure, i.e.,
mapped to themselves

  link_extendedConstant(O,C) :- link_constantInTheory(O,C),
  \begin{align*}
  & I_{\text{extendedConstants}}(O, O).
  \end{align*}
• Common axioms for BAT are useful for later definition of \textit{is\_a} relation, e.g., the definition of source sorts, subsorts etc.

%- 2-1 For every state constraint or definition that involve static functions only, include ASP rules for it. (These rules correspond to axioms 2.2 and 2.3.)
%- 2-2 for every static definition in structure of \( P \), include ASP rules for them.

• Define \textit{is\_a}. Intuitively, \textit{is\_a} is a relation relates any universe element to a source sort such that all object sort links can be “derived”.

%- 2-1: if \((o,c)\) is an “extended” link and \( c \) is source sort.

\[
\text{is\_a}(O,C) :- \text{link\_extendedConstant}(O,C), \text{source}(C).
\]

%- 2-2: if \((o,c)\) is in “extended” link and \( c \) is NOT a source sort. We introduce \( o \) into any leaf (source) subsort of \( c \).

\[
1\{\text{is\_a}(O,\text{SubC}) : \text{subsort}(\text{SubC},C), \text{source}(\text{SubC})\}1 :- \\
\text{link\_extendedConstant}(O,C), \text{not source}(C).
\]

2.2.2.5 Pre-model Defined by a System be Description

Given a system description \( P \), we call the program \( \Pi_A = \Pi_1 \cup \Pi_2 \), where \( \Pi_1 \) and \( \Pi_2 \) are defined as above, the \textit{ASP program defined by} \( P \). Given \( P \), let \( T = (\Sigma = \langle C,O,H,F \rangle, A) \) be the core BAT defined by the theory of \( P \). \( \Sigma \) is called the \textit{signature} defined by \( P \).

\textbf{Definition 13} (Universe and Extended Signature). A \textit{universe} \( U \) defined by an answer set \( S \) of \( \Pi \) is \( U = \{ x : \text{universe}(x) \in S \} \). An \textit{extended signature} \( \langle C,O',H',F \rangle \) defined by an answer set \( S \) of \( \Pi \) is

- Extended Constants: \( O' = \{ o : \text{extendedConstants}(o) \in S \} \),

- Sort hierarchy \( H' = (V,E) \) where \( V = O' \cup C \), \( E = \{ (c_1,c_2) : \text{link}(c_1,c_2) \in S \} \cup \{ (o,c) : \text{link\_extendedConstant}(o,c) \in S \} \).
Clearly, the extended signature defined by an answer set of $\Pi$ is a sorted signature. Let $U$ and $\Sigma' = \langle C, O', H', F \rangle$ be the universe and extended signature defined by an answer set of $\Pi$. Note that $\Sigma'$ and $\Sigma_U$ share the same object constants. The difference between $\Sigma'$ and $\Sigma_U$ is the difference between $H'$ and $H$. Intuitively compared with $\Sigma_U$, $\Sigma'$ includes information on which constant of $U$ belongs to which sort in its sort hierarchy (the sort hierarchy of $\Sigma_U$ contains information only on constants of $O$ but not $U$).

**Definition 14.** The static interpretation $M$ defined by an answer set $S$ of $\Pi$ is

- The universe of $M$ is the one defined by $S$.
- The assignment $M$ is defined as follows:
  - For every $c \in C_{ud}$, $M(c) = \{x : \text{instance}(x, c) \in S\}$.
  - For every $o_1$ such that $\text{extendedConstants}(o_1) \in S$, $M(o_1) = o_2$ such that $I_{\text{extendedConstants}}(o_1, o_2) \in S$.\(^\text{2}\)
  - Each static user defined function $f : c_1 \times \cdots \times c_n \to c_0 \in F_{ud}$ is mapped into the possibly partial function $M(f) : M(c_1) \times \cdots \times M(c_n) \to M(c_0)$ such that, $M(f)(M(t_1), \ldots, M(t_n)) = M(t)$ where $f(t_1, \ldots, t_n) = t \in S$.
  - The special function $\text{is}_a$ is mapped to the function $M(\text{is}_a) : M(\text{is}_a)(o, c)$ is true iff $\text{is}_a(o, c) \in S$.
  - The special function $\text{link}$ is mapped to function $M(\text{link}) : M(\text{link})(c_1, c_2)$ is true iff $\text{link}(c_1, c_2) \in S$.
  - For every static function $f \in F_{ud}$, the special function $\text{dom}_f : c_1 \times \cdots \times c_n \to \text{booleans}$ is mapped to $M(\text{dom}_f)$ such that for every $\bar{x} \in M(c_0) \times \cdots \times M(c_n)$, $M(\text{dom}_f)(\bar{t})$ is true iff $\text{dom}_f(\bar{t}) \in S$.

\(^\text{2}\)This is well defined: $o_2$ exists and unique.
• On predefined symbols, \( M \) is identified with the symbol’s standard interpretation.

\[ \square \]

**Proposition 2.** Let \( \Sigma \) be the signature defined by \( P \) and \( U \) the universe defined by an answer set \( S \) of \( \Pi \). A static interpretation \( M \) defined by \( S \) is a static interpretation of \( \Sigma_U \).

Proof. Let \( \Sigma' = (C',O',H',F') \) be the extended signature defined by \( S \). We will first show that \( M \) is a static interpretation of \( \Sigma' \), and then show that it is a static interpretation of \( \Sigma_U \). Clearly \( M \) consists of only static information.

By the definition of interpretation (definition 1), \( M \) satisfies the clauses in definition 1 in a straightforward manner except the conditions on \( \text{is}(x,c) \) and \( \bigcup \{ M(c_i) : c_i \text{ is a child of } c \} \). Here we use the new definition of interpretation by Gelfond [23].

1) We prove the condition on \( \text{is}(x,c) \). For \( \text{is}(x,c) \), we can verify \( \text{is}(x,c) : \text{universe} \times \text{nodes} \rightarrow \text{boolean} \). Note that \( \text{universe} \) denotes \( U \). We will show that for every \( x \in \text{universe} \) and every sort \( c \in \text{nodes} \), \( \text{is}(x,c) \) is true iff \( c \) is a source node of \( H \) and \( x \in M(c) \).

By definition 14, \( \text{is}(x,c) \) is true iff \( \text{is}(x,c) \in S \). We have two rules of \( \Pi \) defining \( \text{is}(x,c) \):

\[
\text{is}(O,C) : \text{link}\_\text{extendedConstant}(O,C), \text{source}(C).
\]

\[
1\{\text{is}(O,SubC) : \text{subsort}(SubC,C), \text{source}(SubC)\}1 : \\
\text{link}\_\text{extendedConstant}(O,C), \text{not source}(C).
\]

\[ \implies \] Since \( \text{is}(x,c) \in S \), from the the two rules above, we have \( \text{source}(c) \in S \) which implies that \( c \) is a source node (because of the axioms 2.7). Since \( \text{is}(x,c) \in S \), axioms 2.7 implies that \( \text{instance}(x,c) \in S \) which implies that \( x \in M(c) \) by definition 14. Done.

30
⇐: Since $x \in M(c)$, by definition 14, $\text{instance}(x, c) \in S$. Since $c$ is a source sort, there is no $c_1$ such that $\text{link}(c_1, c)$. Hence, $\text{instance}(x, c) \in S$ is supported by the ASP rule of the first axiom in equation 2.7.

Therefore, $\text{is}_{\mathcal{A}}(x, c) \in S$. Done.

2) On $M(c) = \bigcup \{M(c_i) : c_i \text{ is a child of } c\}$. We know $M(c) = \{x : \text{instance}(x, c)\}$ and for each child $c_i$ of $c$, $M(c_i) = \{x : \text{instance}(x, c_i)\}$. Since $c_i$ is a child of $c$ in $H'$, $\text{link}(c_i, c) \in \Pi$. By hierarchy axioms 2.7, $\text{instance}(c)$ is only defined by $\text{instance}(\ast, c_i)$. Therefore, $M(c) = \bigcup \{M(c_i) : c_i \text{ is a child of } c\}$.

Finally, $\Sigma_U$ differs from $\Sigma'$ only by the sort hierarchy where the edges of latter is a super set of the former. Hence, an interpretation of $\Sigma'$ is an interpretation of $\Sigma_U$. Therefore, $M$ is a static interpretation of $\Sigma_U$. \hfill \Box

**Definition 15.** A static interpretation defined by a system description $P$ is a static interpretation defined by an answer set of the program defined by $P$. \hfill \Box

**Theorem 1.** Given a system description $P$, let signature $\Sigma$ and core BAT $T$ be defined by $P$. Let $S$ be an answer set of $\Pi$, $U$ and $M$ are the universe and static interpretation defined by $S$ respectively. $M$ is a pre-model of $T$ with universe $U$.

Proof. Let $\Sigma$ be $(O, C, H, F)$ and $M$ be a static interpretation of $P$. $\Sigma_U = (O \cup U, C, H, F)$. $O$ is the set of declared constants and predefined constants. We can ignore predefined constants.

By definition 2 of pre-models, to show $M$ is a pre-model of $T$ with universe $U$, we show 1) $M$ is a static interpretation of $\Sigma_U$, 2) $M(\text{universe}) = U$, and 3) for every object constant $o$ of $\Sigma_U$ that is not an object constant of $\Sigma$, $M(o) = o$.

1) holds by Proposition 2.

2) holds directly by the first clause of definition 14 of static interpretation defined by an answer set of $\Pi$. 

31
3) for every object constant \( o \) of \( \Sigma_U \), if \( o \notin O \), we show \( M(o) = o \). Since \( o \notin O \), we do not have the following rule in \( \Pi \)

\[
\text{link} \_\text{constantInTheory}(o, c).
\]

Since \( \Pi \) has only one rule defining \( \text{constantInTheory}() \)

\[
\text{constantInTheory}(X) :- \text{link} \_\text{constantInTheory}(X, C)
\]

we have \( \text{constantInTheory}(o) \notin S \).  

(1)

Since \( o \in O \cup U \), we have \( o \in U \) and thus \( \text{universe}(o) \in S \). Since \( \Pi \) has the following rule

\[
I \_\text{extendedConstants}((Id, Id) :- \text{universe}(Id)
\]

we have \( I \_\text{extendedConstants}(o, o) \in S \).  

(2)

By \( \Pi \), the rules defining \( \text{universe}() \) are

\[
\text{universe}(X) :- \text{link} \_\text{constantInStructure}(X, C).
\]

\[
\text{universe}(X) :- \text{constantInTheory}(X), \not\text{consInTheoryDef}(X).
\]

Therefore, \( \text{link} \_\text{constantInStructure}(o, C) \in S \) for some sort \( C \) because of (1). By the following rule of \( \Pi \)

\[
\text{extendedConstants}(X) :- \text{link} \_\text{constantInStructure}(X, C)
\]

we have \( \text{extendedConstants}(o) \in S \), which, together with (2), implies that \( M(o) = o \) by (the second subclause of the second clause of) definition 14. Now we complete the proof of 3) and thus the proof of this theorem.

\[\square\]

2.2.2.6 Semantics of a System Description

**Definition 16.** (Model of System Description)

Given a system description \( P \), let \( \Pi \) be the program defined by \( P \) and \( T \) the core BAT defined by \( P \), and \( U \) and \( M \) the universe and static interpretation defined by an answer set of \( \Pi \). A **model** of \( P \) is a transition diagram \( T_M \) defined by pre-model \( M \) of \( T \) with universe \( U \) such that it has a non-empty collection of states.

\[\square\]
2.3 Design Of CALM

2.3.1 Background

2.3.1.1 SPARC

SPARC is a variant of ASP where the signature of the logic program is explicitly described through providing a formal description of the sort hierarchy and providing the sorted signature of each predicate used in the rules of the program.

Sort Definitions   The sorts section occurs at the top of a SPARC program, begins with the keyword sorts and is followed by a sequence of sort definitions. Each sort definition has the following form:

\[
\text{sort} \text{name} = \text{sort} \text{expression}.
\]

The complete syntax for describing sorts is available in the SPARC manual. We provide here the subset of syntax we use in CALM. Basic sort expressions include enumerated sets of ground terms \{id\textsubscript{1}, id\textsubscript{2}, ...\}, integer ranges number\textsubscript{1}..number\textsubscript{2}, sort names #sort\textsubscript{name}, and records record\textsubscript{name}(#sort\textsubscript{1}, #sort\textsubscript{2}, ..., #sort\textsubscript{n}). Complex sort expressions incorporate set theoretic operators between simple sort expressions. In our case we employ the + operator to perform union operations. A sort must be defined before use in records and complex sort expressions.

Example Sort Section:

```
sorts
    #fruits = \{apples, bananas\}.
    #vegetables = \{peas, carrots\}.
    #edibles = #fruits + #vegetables
        + pair(#fruits,#vegetables).
```

Predicate Declarations   The predicates section follows the sorts section, begins with the keyword predicates and is followed by a sequence of predicate declarations. Each
predicate declaration has the following form:

\[
\text{predicate}_\text{name}(\#\text{sort}_1, \#\text{sort}_2, \ldots, \#\text{sort}_n)
\]

Example Predication Section:

\begin{verbatim}
predicates
prefer(#edibles, #edibles)
eat(#edibles)
available(#edibles)
satiated()
\end{verbatim}

Program Rules The rules section follows the predicate section, begins with the keyword rules and is followed by a sequence of ASP rules over the predicates that have been declared in the predicates section. A rule has the following form:

\[
l_1 \text{ or } l_2 \text{ or } \ldots \text{ or } l_k \leftarrow l_{k+1}, \ldots, l_m, \neg l_{m+1}, \ldots, \neg l_n.
\]

For \( i \in [1..n] \), each \( l_i \) is either a predicate, its negation, or a relation over terms or variables that have occurred within a predicate in the same rule.

Example Rules Section:

\begin{verbatim}
rules
satiated :- eat(X).
eat(X) :- prefer(X,Y), available(X), not -eat(X).
-eat(X) :- prefer(Y,X), available(Y), available(X).
-eat(X) :- not available(X).
available(pair(X,Y)) :- available(X), available(Y).
prefer(apples, bananas). prefer(peas, carrots).
prefer(pair(apples, carrots), apples).
available(apples). available(bananas).
available(carrots).
\end{verbatim}
A solver will ground rules, substituting values for variables in accordance with the predicate signatures and the sort definitions provided in their respective sections.

2.3.2 Translation

2.3.2.1 Algorithm

The steps of the algorithm are explained in detail in the following sections.

CALM Algorithm

\begin{itemize}
\item[] input: an ALM system description $P$
\item[] output: a SPARC program $\Pi_C$
\end{itemize}

1. Parse $P$ to verify that it is syntactically and semantically correct and report any errors.
2. Construct the Core BAT $T$ as defined by $P$.
3. Construct the static SPARC Program $\Pi_M$ from $T$.
4. Obtain the answer sets $A_M$ of $\Pi_M$.
   4.1 There must be exactly one answer set in $A_M$.
5. Construct SPARC program $\Pi_C$ from $\Pi_M$, $A_M$, and $T$.
   5.1 Each sort $c$ is defined by $\{X: \text{instance}(X,c) \in A_M\}$.
   5.2 The predicates section is copied from $\Pi_M$.
   5.3 The rules are copied from $\Pi_M$ and extended with rules translated from the non-static axioms in $T$.

We break the creation of the output program $\Pi_C$ into two parts. First we construct the static program $\Pi_M$ which encodes the action signature $\Sigma_M$ where $M$ is the model defined by $P$. $\Pi_M$ also contains all static axioms and static function definitions which may influence the definition of instances for sorts. If more or less than one answer set exists for $\Pi_M$, we report an error. If there is no answer set to the static program,
then there is no possible transition diagram described by $\Pi_C$. It is more informative to know that the sub-program $\Pi_M$ failed than trying to discern this fact from $\Pi_C$ failing to have an answer set. Currently $\textit{CALM}$ does not support system descriptions which define sort instance for non-source sorts in the hierarchy. Such programs have multiple answer sets for the static program $\Pi_M$.

From our calculation of the answer set $A_M$, we replace the definitions of the sorts in $\Pi_C$ with an enumeration of the instances indicated in $A_M$. We believe the explicit enumeration of each sort with its calculated instances helps to reduce the size of the final ground program and saves some of the computation needed to calculate the sort instances in $\Pi_C$.

Part of our selection of $\textit{SPARC}$ as a target for translation is its ability to represent the sort hierarchy separate from the rules of the program. Its automated type checking of rules based on predicate signatures has proved useful in catching errors during development of $\textit{CALM}$. While the current implementation of $\textit{SPARC}$ translates to the DLV $\textit{ASP}$ solver, a future direct implementation of $\textit{SPARC}$ can utilize the sort definitions and predicate signatures to optimize grounding and solver execution. If happens, $\textit{CALM}$ will benefit from its use of $\textit{SPARC}$ over un-sorted $\textit{ASP}$ variants.

2.3.2.2 Translation of BAT to SPARC

Let a basic action theory $T = (\Sigma = \langle C, O, H, F \rangle, A)$ be given. We construct the SPARC Program $\Pi_C$ in two stages. We first construct the SPARC Program $\Pi_M$ derived from the signature and static rules of $T$.

**Definition 17.** Static Translation $\Pi_M$ of a System Description $P$ Let $T = (\Sigma = \langle C, O, H, F \rangle, A)$ be the core BAT of System Description $P$.

Each sort definition in the sort section of $\Pi_M$ has the following structure:

$$\#\text{sort} = \#\text{sort}_1 + \ldots + \#\text{sort}_n$$
such that the following properties hold:

- All sorts used on the right hand side of the = have been previously defined above this sort definition.

- for $i \in [1..n]$, $\text{sort}_i \in \{\text{sort}_1 \ldots \text{sort}_n\}$ if and only if $(\text{sort}, \text{sort}_i) \in H$.

- for $j \in [1..k]$, $\text{id}_j(\#\text{sort}_{j,1}, \ldots, \#\text{sort}_{j,n_j})$ is in the sort definition if and only if there is a schema definition in the structure of $P$ of the form

$$\text{id}_j(V_{j,1}, \ldots, V_{j,n_j}) \text{ in sort (where } l_1, \ldots, l_m)? \quad f_1(t_1) = V_1 \quad \ldots \quad f_{n_j}(t_{n_j}) = V_{n_j}$$

such that the range sorts of $f_1 \ldots f_{n_j}$ are $\text{sort}_{j,1}, \ldots, \text{sort}_{j,n_j}$ respectively.

- $\text{const} \in \{\text{const}_1 \ldots \text{const}_s\}$ if and only if $\text{const}$ is a ground term and either is a declared constant for $\text{sort}$ in $P$, defines a declared constant for $\text{sort}$ in $P$, or there is an instance definition in $P$ of the form

$$\text{const} \text{ in sort}.$$ 

Each predicate definition in the predicate section of $\Pi_M$ has the following form:

$$\text{fun}(\#\text{sort}_1, \ldots, \#\text{sort}_{n-1}, \#\text{sort}_n)$$
such that either \texttt{fun:sort}_1,\ldots,\texttt{sort}_{n-1} \rightarrow \texttt{sort}_n is a static function in \emph{F} or \texttt{fun:sort}_1,\ldots,\texttt{sort}_{n-2} \rightarrow \texttt{sort}_{n-1} is a fluent function in \emph{F} and \texttt{sort}_n is the time step sort.

The rules section of \Pi_M include all static auxiliary rules (2.2, 2.3, 2.7) and all the static user defined axioms from the theory and the structure translated to \emph{SPARC} rules.

\hspace{1cm} \Box

2.3.2.3 Correctness Of Translation

\textbf{Definition 18. Temporal Interpretation}

Given a BAT signature \Sigma, let \emph{I} be an interpretation of \Sigma and \( T \in \mathcal{N} \). The \emph{temporal interpretation} \( I(T) \) is the set of fluents and statics:

\[
\{ f(\bar{t}, T) = V \mid \text{fluent } f(\bar{t}) = V \in I \} \bigcup \{ f(\bar{t}) = V \mid \text{static } f(\bar{t}) = V \in I \}
\]

\hspace{1cm} \Box

\textbf{Definition 19. Predicate Representation of an Interpretation}

Let \emph{I} be an interpretation for a BAT signature \Sigma and \( T \in \mathcal{N} \). The \emph{predicate representation} of \emph{I} is the set of predicates:

\[
\{ f(\bar{t}, V) \mid f(\bar{t}) = V \in I \} \bigcup \{ \neg f(\bar{t}, V') \mid f(\bar{t}) = V \in I \text{ and } V \neq V' \}
\]

where \( V \) and \( V' \) are different instances in the range sort of \( f \).

\hspace{1cm} \Box

\textbf{Definition 20. State Defined By The \emph{SPARC} Program \Pi_C}

Given a System Description \emph{P} such that \Pi_A, the \emph{ASP} program defined by \emph{P}, has only one answer set \( S \), let \Sigma and \emph{M} be the core BAT and static interpretation
defined by $P$ respectively. Let $\Pi_C$ be the output of $\text{CALM}$ on $P$. An interpretation $\sigma$ of $\Sigma$ with static part $M$ is a state defined by the $\text{SPARC}$ program $\Pi_C$ if and only if $\sigma'(0)$, the predicate representation of the temporal interpretation $\sigma(0)$, is the only answerset of the program $\Pi_C \cup K$ where $K$ is $\sigma'(0)$ with the defined literals removed.

□

**Definition 21.** Transition Defined By The $\text{SPARC}$ Program $\Pi_C$

Given a System Description $P$ such that $\Pi_A$, the $\text{ASP}$ program defined by $P$, has only one answer set $S$, let $\Sigma$ and $M$ be the core BAT and static interpretation defined by $P$ respectively. Let $\Pi_C$ be the output of $\text{CALM}$ on $P$. Let $\sigma_0$ and $\sigma_1$ be states defined by $\Pi_C$ and $a \subseteq M\text{(actions)}$. Let $\sigma_0'(0)$ and $\sigma_1'(1)$ be the predicate representations of the temporal interpretations $\sigma_0(0)$ and $\sigma_1(1)$ respectively. Let $A = \sigma_0'(0) \cup \{\text{occurs}(x, 0)|x \in a\} \cup \sigma_1'(1)$. The triple $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition defined by the $\text{SPARC}$ program $\Pi_C$ if and only if $A$ is the only answer set of the program $\Pi_C \cup K$ where $K$ is $\sigma_0'(0) \cup \{\text{occurs}(x, 0)|x \in a\}$ with the defined literals removed.

□

**Definition 22.** State Transition Diagram Defined By The $\text{SPARC}$ Program $\Pi_C$

Given a System Description $P$ such that $\Pi_A$, the $\text{ASP}$ program defined by $P$, has only one answer set, let $\Pi_C$ be the output of $\text{CALM}$ on $P$. The Transition Diagram defined by $\Pi_C$ is the transition diagram defined by the set of states and transitions defined by $\Pi_C$ if the set of states is non-empty.

□

**Proposition 3.** Given an $\text{ALM}$ system description $P$, let $\Pi_C$ be the output of $\text{CALM}$ on input $P$. $ST$ is a transition diagram defined by $\Pi_C$ if and only if $ST$ is a model of $P$.

**Proof**
Let $\Pi$ be the $\text{ASP}$ program defined by $P$ and $T$ the core BAT defined by $P$, and $U$ and $M$ the universe and static interpretation defined by an answer set of $\Pi$.

Assumption: $ST$ is a transition diagram defined by $\Pi_C$

We will show that $ST$ is a model of $P$ by showing that $\sigma$ is a state of $ST$ if and only if $\sigma$ is a state of $T_M$ and that $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition of $ST$ if and only if $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition of $T_M$.

Assumption: $\sigma$ is a state of $ST$.

$\sigma'(0)$, the predicate representation of the temporal interpretation $\sigma(0)$ with the defined literals removed, is the only answer set of the program $\Pi_C \cup \sigma'(0)$ with the defined literals removed.

Note that $\sigma'(0)$ contains no occurrences of actions.

The rules within $\Pi_C$ that have been translated from dynamic causal laws and executability conditions do not have their bodies satisfied.

The ground $\text{SPARC}$ program $\Pi_C$ with the dynamic causal laws and executability conditions removed is equivalent to the logic program $S_M$.

Since $\sigma'(0)$ is the predicate representation of the temporal interpretation $\sigma(0)$, $\sigma$ is the only answer set of the logic program $S_\sigma$.

$\sigma$ is a state of $T_M$.

Assumption: $\sigma$ is a state of $T_M$.

$\sigma$ is the only answer set of the logic program $S_\sigma$. 
The ground SPARC program \( \Pi_C \) with the dynamic causal laws and executability conditions removed is equivalent to the logic program \( S_M \).

\( \sigma'(0) \), the predicate representation of the temporal interpretation \( \sigma(0) \) with the defined literals removed, is the only answer set of the program \( \Pi_C \cup \sigma'(0) \) with the defined literals removed.

\( \sigma \) is a state of \( ST \).

Therefore \( \sigma \) is a state of \( ST \) if and only if \( \sigma \) is a state of \( T_M \).

Assumption: \( \langle \sigma_0, a, \sigma_1 \rangle \) is a transition of \( ST \)

Let \( A_{ST} = \sigma'_0(0) \cup \{\text{occurs}(x, 0) \mid x \in a\} \cup \sigma'_1(1) \) where \( \sigma'_0(0) \) and \( \sigma'_1(1) \) are the predicate representations of the temporal interpretations \( \sigma_0(0) \) and \( \sigma_1(1) \) respectively.

\( A_{ST} \) is the only answer set of the program \( \Pi_C \cup K \) where \( K \) is \( \sigma'_0(0) \cup \{\text{occurs}(x, 0) \mid x \in a\} \) with the defined literals removed.

Program \( P(M, \sigma_0, a) \) has an answer set \( A \) such that \( f(t_1, \ldots, t_n) = t \in \sigma_1 \), iff \( f(t_1, \ldots, t_n) = t \in A \) when \( f \) is an attribute or static and \( f(t_1, \ldots, t_n, 1) = t \in A \) when \( f \) is a fluent.

\( \langle \sigma_0, a, \sigma_1 \rangle \) is a transition of \( T_M \).

Assumption: \( \langle \sigma_0, a, \sigma_1 \rangle \) is a transition of \( T_M \)

Program \( P(M, \sigma_0, a) \) has an answer set \( A \) such that \( f(t_1, \ldots, t_n) = t \in \sigma_1 \), iff \( f(t_1, \ldots, t_n) = t \in A \) when \( f \) is an attribute or static and \( f(t_1, \ldots, t_n, 1) = t \in A \) when \( f \) is a fluent.

\( \langle \sigma_0, a, \sigma_1 \rangle \) is a transition of \( ST \)
Therefore $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition of $ST$ if and only if $\langle \sigma_0, a, \sigma_1 \rangle$ is a transition of $T_M$

Since $ST = T_M$ and $ST$ has a non-empty set of states,

$ST$ is a transition diagram defined by pre-model $M$ of $T$ with universe $U$ such that it has a non-empty collection of states.

Conclusion: $ST$ is a model of $P$

2.3.3 Semantic Errors

One of the primary contributions of $CALM$ is the integration of syntax and semantic error detection in the compilation process of translating a System Description to the $SPARC$ program $\Pi_C$. In this section we encode the understanding of a Well Defined System Description. We detail the semantic errors in context of their relevant grammatical elements from the syntax of a system description.

When evaluating a grammatical element for semantic errors it is evaluated with respect to the partial construction of the Basic Action Theory (BAT) $\langle C, O, H, F \rangle$ defined by processing module dependencies and previously processed grammatical elements.

2.3.3.1 Theory

The following semantic requirements must be satisfied for locally defined theories:

- Within a System Description there is exactly one theory defined or imported.
- An imported theory must indicate the library from which it is being imported and the library and theory name must resolve to a file on disk.
• Within a library file, there is exactly one theory defined and the theory name
matches the containing file.

Module Declarations  The following semantic requirements must be satisfied for mod-
ule declarations and import statements.

• Module import statements must indicate the name of the containing library and
theory and the name of the module being imported.

• Module import statements must resolve to a library and theory file on disk
which contains a locally defined module of the same name being imported.

• The names of modules which are locally defined or directly imported must be
unique within the context of the containing theory.

• A module dependency declaration must resolve to either a locally defined mod-
ule or directly imported module in the containing theory.

Sort Declarations  For a sort declaration statement of the form

\[id_1, \ldots, id_n :: sort_1, \ldots, sort_m\]
\[attr_f : c_1 \times \cdots \times c_k \to c_0\]

The following semantic requirements must be satisfied:

• For \( j \in [1..n] \), each \( sort_j \) must have been previously declared and exists as a
node in \( H \). (Note that this is sufficient to guarantee a path exists in \( H \) from
the node labeled by \( sort_j \) to the node labeled by \( universe \).)

• For \( i \in [1..n], j \in [1..m] \), each \( id_i \) and \( sort_j \) must be a unique string.

• For \( i \in [1..n], j \in [1..m] \), each \( id_i \) must not occur along any path from the node
labeled by \( sort_j \) to the node labeled by \( universe \) in \( H \).
• For \( i \in [1..n], j \in [1..m] \), if a node labeled by \( id_i \) exists in \( H \), there must not already be an arc in \( H \) from the node labeled by \( id_i \) to the node labeled by \( sort_j \).

• For \( i \in [1..n], j \in [1..m] \), add \( id_i \) to \( C \) and nodes labeled by \( id_i \) to \( H \) if they do not exist, and add arcs from \( id_i \) to \( sort_j \) in \( H \).

• For \( l \in [0..k] \), there must be a node labeled by \( c_l \) in \( H \).

• For \( i \in [1..n] \), there must not already be a function \( attr_f : id_i \times c_1 \times \cdots \times c_k \rightarrow c_0 \) in \( F \).

• For \( i \in [1..n] \), add the function \( attr_f : id_i \times c_1 \times \cdots \times c_n \rightarrow c_0 \) to \( F \) with markers indicating the function is static and basic.

Constant Declarations  
For a constant declaration statement of the form

\[
  id_1, \ldots, id_n :: sort_1, \ldots, sort_m
\]

The following semantic requirements must be satisfied:

• For \( i \in [1..n] \), \( id_i \) must not be a string in either \( C \) or \( O \).

• For \( j \in [1..m] \), \( sort_j \) must be in \( C \).

• For \( i \in [1..n] \), add \( id_i \) to \( O \) and add a node labeled by \( id_i \) to \( H \).

• For \( i \in [1..n], j \in [1..m] \), add an object constant arc from the node labeled by \( id_i \) to the node labeled by \( sort_j \) in \( H \).

Function Declarations  
For a function declaration statement of the form

\[
  \texttt{[total]} \ f : sort_1, \ldots, sort_n \rightarrow sort_0
\]

The following semantic requirements must be satisfied:
• If \( f \) is a defined function \( \text{sort}_0 \) must be \textit{booleans}.

• For \( i \in [0..n] \), \( \text{sort}_i \) must be in \( C \).

• \( f : \text{sort}_1, \ldots, \text{sort}_n \rightarrow \text{sort}_0 \) must not already be a function in \( F \).

• Add \( f : \text{sort}_1, \ldots, \text{sort}_n \rightarrow \text{sort}_0 \) to \( F \) with markings indicating whether or not the function is total, static or fluent, and basic or defined.

• Add \( \text{dom} f : \text{sort}_1, \ldots, \text{sort}_n \rightarrow \text{booleans} \) to \( F \) with markings indicating whether or not the function is total, static or fluent, and basic or defined.

Dynamic Causal Laws  For a dynamic causal law of the form

\[
\text{occurs}(a) \text{ causes } f(x_1, \ldots, x_n) = o \text{ if } \text{instance}(a, c), \text{ cond}
\]

The following semantic requirements must be satisfied:

• \( a \) must be a variable.

• \( c \) must be in \( C \) and a subsort of \textit{actions} according to \( H \).

• \( f \) must be a basic fluent function in \( F \).

• \( o \) must be either a variable occurring in the conditions, a constant in \( O \), or a function whose inferred sort is compatible with the range sort of \( f \).

• For \( i \in [1..n] \), \( x_i \) must be either a variable, a constant in \( O \), or a function and the inferred sort of \( x_i \) must be compatible with the \( i_{th} \) domain sort in the signature of \( f \).

• The type inferred for each variable in this axiom must resolve to a sort in the sort hierarchy.
• All functions occurring in the conditions of this axiom must have a unique signature in \( F \) and have agreement between their signature and the inferred sort for the syntactic elements appearing as domain arguments and the range value of the function.

Executability Conditions  For an executability condition of the form

\[
\text{impossible occurs}(a) \text{ if } \text{instance}(a, c), \text{ cond}
\]

The following semantic requirements must be satisfied:

• \( a \) is a variable.

• \( c \) must be in \( C \) and a subsort of \( \text{actions} \) according to \( H \).

• The type inferred for each variable in this axiom must resolve to a sort in the sort hierarchy.

• All functions occurring in the conditions of this axiom must have a unique signature in \( F \) and have agreement between their signature and the inferred sort for the syntactic elements appearing as domain arguments and the range value of the function.

State Constraints  For state constraints of the form

\[
f(x_1, \ldots, x_n) = o \text{ if } \text{cond}
\]

The following semantic requirements must be satisfied:

• \( f \) must be a basic function in \( F \).

• If \( f \) is a static function then only static functions may occur in the conditions of the state constraint.
• $o$ must be either a variable occurring in the conditions, a constant in $O$, or a function whose inferred sort is compatible with the range sort of $f$.

• For $i \in [1..n]$, $x_i$ must be either a variable, a constant in $O$, or a function and the inferred sort of $x_i$ must be compatible with the $i$th domain sort in the signature of $f$.

• The type inferred for each variable in this axiom must resolve to a sort in the sort hierarchy.

• All functions occurring in the conditions of this axiom must have a unique signature in $F$ and have agreement between their signature and the inferred sort for the syntactic elements appearing as domain arguments and the range value of the function.

Function Definitions For function definitions of the form

$$f(x_1, \ldots, x_n) \ if \ cond$$

The following semantic requirements must be satisfied:

• $f$ must be a defined function in $F$.

• If $f$ is a static function then only static functions may occur in the conditions of the function definition.

• For $i \in [1..n]$, $x_i$ must be either a variable, a constant in $O$, or a function and the inferred sort of $x_i$ must be compatible with the $i$th domain sort in the signature of $f$.

• The type inferred for each variable in this axiom must resolve to a sort in the sort hierarchy.
• All functions occurring in the conditions of this axiom must have a unique signature in $F$ and have agreement between their signature and the inferred sort for the syntactic elements appearing as domain arguments and the range value of the function.

2.3.3.2 Structure

All semantic requirements of a structure are related to how its syntactic expression extend the core BAT of the theory.

Constant Definitions  For constant definitions of the form

\[ \text{const} = \text{groundterm} \]

The following semantic requirements must be satisfied:

• \text{const} must be a declared constant in $O$.

• \text{const} must not have been defined by another constant definition statement.

• This constant definition is recorded for later reference.

• The \text{groundterm} is recorded as an instance of every sort $c$ for which there is an object constant arc from the node \text{const} to a node labeled by $c$ in $H$.

Instance Definitions  For instance definitions of the form

\[ \text{groundterm in c} \]

The following semantic requirements must be satisfied:

• $c$ must be a sort name in $C$ and a label of a node in $H$.

• The \text{groundterm} is recorded as an instance of $c$.

For instance definitions of the form
\[ id(V_1, \ldots, V_n) \text{ in } c \text{ where } l_1, \ldots, l_m \]
\[ f_1(t_{1,1}, \ldots, t_{1,k_1}) = V_1 \]
\[ \ldots \]
\[ f_n(t_{n,1}, \ldots, t_{n,k_n}) = V_n \]

The following semantic requirements must be satisfied:

- \( c \) must be a sort name in \( C \) and a label of a node in \( H \).
- \( id \) must not be the name of a function in \( F \) or a declared constant in \( O \).
- For \( i \in [1..n] \), \( V_i \) is a variable
- For \( i \in [1..n] \), the signature of \( f_i \) is an attribute function \( f_i : c'_i \times c_{i,1} \times \cdots \times c_{i,k_i} \to c_{i,0} \) in \( F \) such that \( c \) is a subsort of \( c'_i \) in \( H \).
- For \( i \in [1..n] \), the inferred sort of \( V_i \) is \( c_{i,0} \), \( c \) must not be a subsort of \( c_{i,0} \), and \( c \) is marked as dependent on \( c_{i,0} \) in the BAT for later reference.
- The type inferred for other variables in this instance schema definition must resolve to a sort in the sort hierarchy.
- All functions occurring in this instance schema definition must have a unique signature in \( F \) and have agreement between their signature and the inferred sort for the syntactic elements appearing as domain arguments and the range value of the function.

Static Function Definitions

Static function definitions in the structure have the same semantic requirements as static function definitions in the theory.
2.3.3.3 Type Checking

For simplicity, our discussion focuses on the type checking of the theory of a system description.

For example, consider an example in the travel domain we discussed earlier. Assume function \( \text{location} : \text{agents} \rightarrow \text{locations} \), and a state constraint

\[
\text{location}(C) = P \text{ if } \text{holding}(A,C), \text{location}(A) = P.
\]

We know function \( \text{holding} : \text{agents} \ast \text{carriables} \rightarrow \text{booleans} \). So, the literal \( \text{holding}(A,C) \) means that the variable \( C \) belongs to sort \( \text{carriables} \). However, the literal \( \text{location}(C) = P \) implies that \( C \) is of sort \( \text{agents} \) by function \( \text{location} \). We also know that the sort names \( \text{carriables} \) and \( \text{agents} \) are not related in terms of subsort. So, we spot an error using sort (type) information of the theory.

We will define below type compliance of a BAT theory. Naturally, the definition is recursive. We first introduce sort compatibility and then start the definition of compliance from terms.

The sorts \( S_1, ..., S_n \) of an action signature are \textit{compatible} if they have a common descendant in the directed graph of the sort hierarchy of the signature.

Given a BAT, a term \( t \) of form \( f(t_1, ..., t_n) \) and a variable occurrence \( X \),

- if \( X \) is \( t_i \), let the sort of \( i^{th} \) parameter of \( f \) be \( s \), \( s \) is the \textit{inferred sort} of \( X \) from \( t \),

- otherwise, let \( X \) occur in \( t_i \) and \( s \) be the inferred sort of \( X \) from \( t_i \), \( s \) is the \textit{inferred sort} of \( X \) from \( t \).

The inferred sorts of variable \( X \) from a term \( t \) is the set of the inferred sort of every occurrence of \( X \) from \( t \).

We can obtain the inferred sort of a constant occurrence (and constant respectively) from the definition of the inferred sort of a variable occurrence (and variable...
respectively) by replacing variables with constants. For simplicity, we assume no constant contains a functor.

Given a BAT, a term \( t \) of form \( f(t_1, \ldots, t_n) \) is compliant with sort \( S \) if

- the range of \( f \) is compatible with \( S \),
- for every \( t_i(i \in 1..n) \), \( t_i \) is compliant with \( s_i \) which is the sort of the \( i^{th} \) parameter of \( f \), and
- for every constant or variable \( x \) of \( t \), the inferred sorts of \( x \) from \( t \) are compatible.

In the following, \( \text{occurs}(a) \), where \( a \) is either a variable or a constant, is also called a literal.

Given a BAT, a literal \( l \) and a variable occurrence \( X \),

- if \( l \) is of the form \( X = f(t_1, \ldots, t_n) \) or \( f(t_1, \ldots, t_n) = X \) and range of \( f \) is \( s \), \( s \) is the inferred sort of \( X \) from \( l \),
- if \( l \) is of the form \( f(t_{11}, \ldots, t_{1m}) = g(t_{21}, \ldots, t_{2m}) \), \( X \) occurs in \( f(t_{11}, \ldots, t_{1m}) \) (or \( g(t_{21}, \ldots, t_{2m}) \) respectively) and the inferred sort of \( X \) from \( f(t_{11}, \ldots, t_{1m}) \) (or \( g(t_{21}, \ldots, t_{2m}) \) respectively), \( s \) is the inferred sort of \( X \) from \( l \),
- if \( l \) is of the form \( \text{occurs}(X) \), the special sort \( \text{actions} \) is the inferred sort of \( X \) from \( l \), and
- if \( l \) is of the form \( \text{instance}(X, s) \) where \( s \) is a sort name, \( s \) is the inferred sort of \( X \) from \( l \).

The inferred sorts of variable \( X \) from a literal is the set of the inferred sorts of every occurrence of \( X \) from \( l \).

Similarly, we define the inferred sorts of constant and constant occurrence from a literal.

Give a BAT, a literal \( l \) is type compliant if
• when $l$ is of form $t_1 = t_2$, or $t_1 \neq t_2$, where $t_1 = f(t_{11}, ..., t_{1m})$ and $t_2 = g(t_{21}, ..., t_{2n})$ are terms, the ranges of $f$ and $g$ are compatible, $t_1$ is compliant with range of $f$ and $t_2$ is compliant with range of $g$,

• when $l$ is of form $X = f(t_1, ..., t_n)$ or $f(t_1, ..., t_n) = X$, $f(t_1, ..., t_n)$ is compliant with the range of $f$,

• for every constant or variable $x$, the inferred sorts of $x$ from $l$ are compatible.

Given a BAT, an axiom is type compliant if

• every literal of the axiom is type compliant, and

• for every constant or variable $x$, the inferred sorts of $x$ from the literals of the axiom are compatible.

A theory is type compliant if its every axiom is type compliant.

Note that due to space limitations, the type checking definition does not include arithmetic relations.

2.3.4 Reasoning System

2.3.4.1 Histories and Trajectories

Definition 23. A Trajectory Of The State Transition Diagram Defined by $\Pi_C$

A sequence $⟨\sigma_0, a_0, \sigma_1⟩, ⟨\sigma_1, a_1, \sigma_2⟩, ..., ⟨\sigma_{n-1}, a_{n-1}, \sigma_n⟩$ of transitions defined by $\Pi_C$ is called a trajectory of the transition diagram of $\Pi_C$.

Note that a trajectory is not required to define range values for every domain value of a basic fluent. Law of inertia for $dom_f$ will preserve the undefined range value until the range value is causally determined by a dynamic causal law.
Definition 24. A History for A System Description

Given an ALM System Description P, a History Π_H for P is a collection of ground facts defined as follows:

- if observed(f(\bar{x}), v, t) ∈ Π_H then f(\bar{x}) is a ground instance of a user defined fluent in P, v is a ground instance in the range of f, and t is a positive integer time step,
- if happened(a, t) ∈ Π_H then a is a ground instance of a subsort of actions in P and t is a positive integer time step,
- there are no other facts in Π_H.

Definition 25. The Complete History Encoding of a Trajectory

Given an ALM System Description P, and a trajectory ⟨σ_0, a_0, σ_1⟩, ⟨σ_1, a_1, σ_2⟩, ..., ⟨σ_n−1, a_{n−1}, σ_n⟩ of the transition diagram of the SPARC program Π_C produced by CALM on P, the complete history encoding of the trajectory is the program Π_H defined as follows:

- observed(f(\bar{x}), v, t) ∈ Π_H if and only if f(\bar{x}) = v ∈ σ_t and f is a basic fluent.
- happened(a, t) ∈ Π_H if and only if a ∈ a_t.

Definition 26. Model Of A History

Given an ALM System Description P with SPARC program Π_C produced by CALM on P and A History Π_H for P, a trajectory T = ⟨σ_0, a_0, σ_1⟩, ⟨σ_1, a_1, σ_2⟩, ..., ⟨σ_{n−1}, a_{n−1}, σ_n⟩ of the transition diagram of Π_C is called a model of Π_H if there exists a program Π_{CH} such that the following properties hold:
• $\Pi_{CH}$ is the complete history encoding of $T$,

• If $\text{observed}(f(\bar{x}), v, t) \in \Pi_H$ then $\text{observed}(f(\bar{x}), v, t) \in \Pi_{CH}$,

• $\text{happened}(a, t) \in \Pi_H$ if and only if $\text{happened}(a, t) \in \Pi_{CH}$.

Note that this definition of model of a history is consistent with the definition in the original $\text{ALM}$ paper[24].

2.3.4.2 Temporal Projection

Definition 27. Temporal Projection Problem

Given an $\text{ALM}$ System Description $P$ with output of $\text{CALM}$ $\Pi_C$ and a history $\Pi_H$ for $P$, a temporal projection problem for $P$ is the $\text{SPARC}$ program $\Pi_{TP}$ such that

• The sort section of $\Pi_{TP}$ is the sort section of $\Pi_C$.

• The predicate section of $\Pi_{TP}$ is an extension of the predicate section of $\Pi_C$ with the following declarations:

  – $\text{observed}(\#\text{universe}, \#\text{universe}, \#\text{timeStep})$

  – $\text{happened}(\#\text{action}, \#\text{timeStep})$

• The rules section of $\Pi_{TP}$ is the rules section of $\Pi_C$ extended by $\Pi_H$ and the following rules:

  – $\text{occurred}(A, I) : \text{happened}(A, I)$.

  – $f(\bar{X}, V, 0) : \text{observed}(f(\bar{X}), V, 0)$.

  for each $\text{observed}(f(\bar{x}), v, 0) \in \Pi_H$.

  – $: \text{not} f(\bar{X}, V, I), \text{observed}(f(\bar{X}), V, I)$. 

54
for each observed\( (f(\bar{x}), V, I) \in \Pi_H \).

□

**Definition 28.** Solution of Temporal Projection Problem \( \Pi_{TP} \)

Given a temporal projection problem \( \Pi_{TP} \) for an \( \mathcal{ALM} \) System Description \( P \), let \( \Pi_C \) be the output of \( \text{CALM} \) on \( P \) and \( \Pi_H \) be the history, the collection of facts, added to \( \Pi_C \) to create \( \Pi_{TP} \). Each model of \( \Pi_H \) is called a *solution of \( \Pi_{TP} \).*

□

Each answer set produced by the \( \text{SPARC} \) solver on a temporal projection problem \( \Pi_{TP} \) encodes a model for the given history.

2.3.4.3 Planning

**Definition 29.** Planning Problem

Given an \( \mathcal{ALM} \) System Description \( P \) and a temporal projection problem \( \Pi_{TP} \), let \( t \) be the maximum time step in the facts of \( \Pi_H \), Let \( \Pi_S \) be a collection of observations of fluent values at \( t' > t \), i.e. \( \Pi_S \subset \{ \text{observed}(f(\bar{x}), v, t') | f \text{ is a basic fluent in } P \} \). The pair \( \langle \Pi_{TP}, \Pi_S \rangle \) is called a *planning problem* for \( P \). \( \Pi_S \) is called the *goal state* of the planning problem.

□

**Definition 30.** Solutions of Planning Problem \( \langle \Pi_{TP}, \Pi_S \rangle \)

Given a planning problem \( \langle \Pi_{TP}, \Pi_S \rangle \) for an \( \mathcal{ALM} \) System Description \( P \), let \( \Pi_C \) be the output of \( \text{CALM} \) on \( P \) and \( \Pi_H \) be the history of facts in \( \Pi_{TP} \) not in \( \Pi_C \), let \( t \) be the maximum time step in \( \Pi_H \) and \( t' \) be the time step of the facts in \( \Pi_S \). Let \( \Pi_A \) be a subset of \( \{ \text{happened}(A, I) | \text{where } A \text{ is a ground instance of an action in } P \text{ and } t < I < t' \} \). \( \Pi_A \) is called a *solution of the planning problem* \( \langle \Pi_{TP}, \Pi_S \rangle \) if the history \( \Pi_H \cup \Pi_A \cup \Pi_S \) has a model and the following properties hold:
• There does not exist a time $k$ such that $\Pi_A$ has no action occurring at time $k$ but has an action occurring after $k$.

• There is no proper subset $\Pi_A'$ of $\Pi_A$ such that $\Pi_H \cup \Pi_A' \cup \Pi_S$ has a model.

Our method of solving planning problems is provided in section 2.4.4.3. We follow method provided in section 4.2.1 of the original $\mathcal{ALM}$ paper[24]. Consistency restoring rules are used to generate actions after the last time step in the history while the goal state is not achieved by the last time step allowed in the horizon. Each answer set of the $\mathcal{SPARC}$ program implementing a planning problem encodes a model of an extension of the provided history with a schedule of actions that achieves the goal state within the horizon.

2.4 Implementation of $\mathcal{CALM}$

Java was selected as the implementation language for its cross-platform portability as a compiled jar.

2.4.1 Parsing System Descriptions and Tasks

2.4.1.1 ANTLR4 Parser

ANTLR4 was selected for its ability to generate a syntax parser with hooks for adding semantic processing to successfully parsed non-terminals. The ANTLR4 grammar for $\mathcal{ALM}$ is in Appendix A. Instructions on how to modify the grammar and change the semantics of $\mathcal{ALM}$ are provided in Appendix B. The output of the parser is an object model for the abstract syntax tree of a system description. The syntax tree of a system description is processed in two passes. The first pass builds a hierarchy of module dependency which may include references to external libraries and theories. The second pass through the syntax processes each module referenced into
a symbol table modeling the BAT structure the module encodes. During this second pass, each $\mathcal{ALM}$ statement is evaluated for semantic errors before being added to the symbol table modeling the BAT structure. Special auxiliary and user defined axioms are added to a collection of $\mathcal{ASP}\{f\}$ rules specifying constraints and requirements of the ontology and state transition diagram. If semantic errors are discovered, they are added to an error report displayed to the user to indicate where in the syntax errors are occurring. If no syntax or semantic errors are generated, the BAT and collected $\mathcal{ASP}\{f\}$ axioms are translated to $SPARC$.

2.4.2 Modeling a Basic Action Theory

Proper semantic evaluation of a System Description and translation to a BAT $(\Sigma = \langle C, O, H, F \rangle, A)$ requires modeling the hierarchy of module dependencies. Our implementation uses a hierarchical symbol table to build a modular BAT representation.

2.4.2.1 Symbol Table

Our symbol table is primarily implemented as several Java based key-value maps. For simple identifiers such as names of constants and sorts, the keys are Strings and the values assigned are the instances of classes designed to model relevant relationships and properties.

To model the set of sort names $C$ we map the string name of the sort to a SortEntry class which has a parent reference and set of child references to model the tree structure of the sort hierarchy $H$. To model links from object constants to sort nodes in $H$, each SortEntry has an initially empty set of instance objects that are populated by instance definitions when parsing the structure of a system description. Initial $universe$ and $actions$ sort entries are created and added to the map.

The set of object constants $O$ in the core BAT is modeled as a map from string
names of constants to instances of a ConstantEntry class which contains a field to indicate any definition of the constant and a set of SortEntry elements to indicate which sorts the constant belongs to. The extended object constants in $O$ of the extended BAT signature are modeled by the set of instances within each SortEntry.

The set of function signatures $F$ is modeled as two maps. The first is a map from function names to sets of normal function signature entries for the name. The second map is from normal function signatures to the related $\text{dom}_f$ signature for the function.

The axioms of $A$ are stored in $\text{ASP}\{f\}$ form and are organized into thematic sections. The sections are ordered when they are translated to $\text{SPARC}$. The auxiliary axioms required to model the sort hierarchy and other BAT constructs are added their own section before parsing a system description.

### 2.4.2.2 Module Hierarchies

In order to properly evaluate the semantic soundness of a module with respect to its module dependencies, a symbol table per module is created. The symbol table for one module has references to the symbol tables of the modules that are included in dependency declarations. If the use of a sort or function in this module does not have a local declaration within the module, they are looked up in the dependent symbol tables recursively. If the use of a sort, function or constant cannot be resolved against declarations through module dependency, semantic errors are reported.

Before translation, all symbol tables are flattened by taking the union of the different maps across the symbol tables in the module hierarchy.

### 2.4.3 Error Checking and Reporting

An ErrorReport singleton class is accumulates syntax errors generated by the parser and passed into the call-back handler for semantic evaluation of successfully
parsed non-terminals in a system description. Error messages are generated for both syntax and semantic errors which indicate file, line and column number of the syntax generating the error.

2.4.3.1 Message Structure

When the ANTLR4 Parser encounters an unexpected symbol in the grammar, it invokes a call-back method we provide to collect syntax errors. The parser provides the location of the error and a RecognitionException object which contains the offending token and a set of expected tokens for that location. We preserve these reported instances in a list of SyntaxError objects collected in the ErrorReport to be displayed after parsing has completed.

Semantic Errors have the following four parts:

1. A unique string called an ErrorId,

2. The message template to display,

3. An explanation of the semantic error,

4. And a recommendation for how to fix the error.

Example:

ErrorID: CND008

Message: Term [graspers(A) at (temp-test.alm:48:32)] has not been declared as a function term or a constant for any sort.

Explanation: Non pre-defined and non-variable terms must be declared before they are used.

Recommendation: Either add a function definition or a constant definition for the term, or replace it
with a declared term.

The semantic error text is stored in a pre-defined table indexed by the unique ErrorId. The message template of each error contains numbered positions that can be filled in dynamically with the syntax tokens that are causing the semantic error to occur.

2.4.3.2 Semantic Error Checking

When non-terminal in the grammar is completed successfully, the ANTLR4 parse calls a method to semantically process the grammatical elements in the non-terminal. Within each non-terminal handler function the grammatical elements are first evaluated for any semantic errors. If a semantic error is found, new semantic error object is created in the ErrorReport with the appropriate ErrorId and a sequence of offending tokens in the grammar to fill in the message template. If no semantic errors are created in the non-terminal handler, the grammatical elements are entered into the appropriate parts of the symbol table.

When displaying semantic errors at the end of parsing the system description, each message template is filled by the sequence of syntax tokens provided at the time of error creation. Each numbered entry in the template is replaced with the corresponding token’s text and file name, line and column number of where the offending token occurs.

2.4.3.3 Type Checking

Each axiom in the theory and each instance definition and static function definition in the structure must pass type checking. Type checking is passed when every variable in the generated logic rules has a concrete inferred type and there is agreement between the expected sorts defined by a function signature and the inferred sorts of the domain arguments and range value of the function.
For example consider the function signature $f : c_1 \times \cdots \times c_n \rightarrow c_0$ and the occurrence of a literal $f(x_1,\ldots,x_n) = x_0$. The expected sort of $x_1$ is $c_1$ from the signature of $f$. If $x_1$ is a constant or instance, then it has a set of sorts $s_1,\ldots,s_m$ for which it has been declared. As long as one of $s_1,\ldots,s_m$ is a subsort of $c_1$, then type checking passes for $x_1$ in the context of $f$. If $x_1$ is a variable, then we add the requirement that the inferred sort of variable $x_1$ be a subsort of $c_1$.

Variable type checking is implemented as follows: For each occurrence of a variable, the required sort is determined as the intersection of all the inferred sorts for the variable’s occurrences. Intersection here is determined as the greatest common subsort of all inferred sorts for the variable.

For example, consider a dynamic causal law for the form:

$\text{occurs}(A)$ causes $f(x_1,\ldots,x_n) = o$ if $\text{instance}(A,c_1)$, $\text{instance}(A,c_2)$, $\text{cond}$

The first occurrence of $A$ has an inferred sort of $\text{actions}$. The second occurrence of $A$ has an inferred sort of $c_1$. The third occurrence of $A$ has an inferred sort of $c_2$. The required sort from these occurrence is $\text{actions} \cap c_1 \cap c_2$, i.e. $\bigcup c_3$ where $c_3$ is a subsort of $\text{actions}$, $c_1$, and $c_2$. If no common subsort exists, then the required sort is labeled as the $\text{EMPTY}$ type. If the variable is free and has no inferred sort, it is labeled with the $\text{ANY}$ type.

The emergent type system here is analogous to expressions in set-calculus. Expressions are reducible under the following reduction rules:

1. $\cap$ and $\cup$ are commutative operators and $\cap$ binds more tightly than $\cup$ as an infix operator.

2. $c_1 \cap c_2 = c_1$ if $c_1$ is a subsort of $c_2$

3. $c_1 \cap c_2 = \text{EMPTY}$ if neither is a subsort of the other.

4. $c_1 \cap \text{EMPTY} = \text{EMPTY}$
5. $c_1 \cap \text{ANY} = c_1$

6. $c_1 \cup c_2 = c_1$ if $c_2$ is a subsort of $c_1$.

7. $c_1 \cup \text{ANY} = \text{ANY}$

8. $c_1 \cup \text{EMPTY} = c_1$

9. $c_1 \cup \cdots \cup c_n = c$ if $c_1, \ldots, c_n$ is a list of all direct subsorts of $c$.

Variable type checking passes when every variable in a rule has a reduced type that is not ANY or EMPTY.

2.4.4 Translation to SPARC

The implementation of translation from BAT to SPARC is performed through creating statements that populate the Sorts, Predicates and Rules section of the produced SPARC program.

2.4.4.1 Sorts Section

The sorts section of the produced SPARC program is produced by recursively defining each sort in the sort hierarchy. Beginning with the universe sort, before a sort is defined, its immediate subsorts and its dependent sorts must first be defined in the output program. The produced SPARC sort definitions have the following form:

$$\#c = \#c_1 + \cdots + \#c_n + s_1(\#c_{1,1}, \ldots, \#c_{1,m_1}) + \cdots + s_k(\#c_{k,1}, \ldots, \#c_{k,m_k}) + \{g_1, \ldots, g_l\}$$

where $c_1, \ldots, c_n$ are the names of sorts that are direct subsorts of $c$ in the sort hierarchy, $s_1(\#c_{1,1}, \ldots, \#c_{1,m_1}), \ldots, s_k(\#c_{k,1}, \ldots, \#c_{k,m_k})$ are derived from schema instance definitions where each $c_{i,j}$ is the inferred sort of the $j_{th}$ variable in the $i_{th}$ schema definition for sort $c$, and $g_1, \ldots, g_l$ are ground instances defined for sort $c$. The SPARC sort definition for $c_1, \ldots, c_n$ and each $c_{i,j}$ in schema pattern for $s_i$ must occur before the SPARC sort definition for $c$. 

62
Special and predefined sort definitions such as ranges of integers and the time steps allowed in trajectories are defined as enumerations of ground terms.

\[ \#timeStep = \{0, 1, 2, 3, \ldots, n\} \]

2.4.4.2 Predicates Section

The predicate section is populated by the predicate signatures modeling the function signatures in the set of functions \( F \) in the BAT. Let \( f : c_1 \times \cdots \times c_n \rightarrow c_0 \) be a function signature in \( F \). If \( f \) is a static function, the predicate signature has the following form:

\[ \text{prefix}_f(#c_1, \ldots, #c_n, #c_0) \]

If \( f \) is a fluent function, the predicate signature has the following form:

\[ \text{prefix}_f(#c_1, \ldots, #c_n, #c_0, \#timeStep) \]

In both cases the prefix is determined by whether or not the function is qualified by additional namespace requirements and whether the function is a special domain function. In the current version of \( \text{CALM} \) the only functions which are namespace qualified are attribute functions. For example if the function is a domain function for an attribute function \( f \) of sort \( c \), the predicate signature would have the following form:

\[ \text{dom}_c.f(#c, #c_2, \ldots, #c_n, #c_0) \]

Future versions of \( \text{CALM} \) may add additional namespace modeling to allow different modules to re-use the same function names locally but have them resolve to different functions globally in the translated program.

2.4.4.3 Rules Section

After parsing the grammatical elements of an \( \text{ALM} \) System Description, the rules modeling the the semantics of the statements are stored in \( \text{ASP}\{f\} \) form in
the symbol table. These rules along with auxiliary rules for modeling the BAT structure and modeling functions as predicates must be translated from $\text{ASP}\{f\}$ to their $\text{SPARC}$ equivalent.

Normalization and Function Translation  All rules, when translated from $\text{ASP}\{f\}$ to their $\text{SPARC}$ equivalent must go through a process of normalization and translation from functions to predicates. $\text{CALM}$ supports nested function terms in the syntax of ALM System Descriptions. $\text{SPARC}$ does not support functions or the ability to nest predicates within each other.

We explain the process of normalization by an example of nested fluent functions:

$$\text{foo}(A, \text{bar}(B)) = \text{baz}(C).$$

The result of normalization is that nested function are replaced by new variables and new literals are added to the end of the body of the rule that equate the variable with the function it replaced. The normalization of the above literal is the following:

$$\text{foo}(A, Z_1) = Z_2, \text{bar}(B) = Z_1, \text{baz}(C) = Z_2.$$

After normalization, the functions can be translated to their predicate equivalent form. Since these functions are fluents, they have a time variable added to their domain.

$$\text{foo}(A, Z_1, Z_2, T), \text{bar}(B, Z_1, T), \text{baz}(C, Z_2, T).$$

Special Auxiliary Rules  There are many rules needed in addition to the explicit user defined axioms provided in the System Description. In order to model the semantics of BAT, the Sort Hierarchy $H$ must be modeled along with domain functions $\text{dom}_f$ in $F$. The axioms for law of inertia on fluents (and their domain functions) and closed world assumption on defined functions must also be added. Since we are modeling
functions as predicates, for every function $f$ in $F$ we must add uniqueness constraints on value assignments to the function. Example:

$$-f(\bar{t}, V_2) : - f(\bar{t}, V_1), \ V_1 \neq V_2.$$ 

All of these auxiliary rules are added to the produced \textit{SPARC} program in their own sections prior to translating any of the user defined axioms in the system description.

Static Program Rules  The user defined static state constraints and static function definitions from the theory and static function definitions from the structure are added to the \textit{SPARC} program in their respective sections.

Once all the static user defined rules are added, we send the static \textit{SPARC} program to the solver to verify that it has a unique answer set. Static programs with multiple answer sets are not currently supported by \textit{CALM}. If there is no answer set, translation halts and a semantic error is reported to indicate that the static program needs to be corrected before fluent rules can be added.

In the current implementation of \textit{CALM} we use the resulting answer set $A$ to replace the sort definitions in the sort section of the \textit{SPARC} program with enumerations from the instance literals in the answer set.

$$\#c = \{x \mid \text{instance}(x, c) \in A\}$$

Our thoughts on this initially is that it would be an optimization step on the grounder to not have to recalculate the sorts in the fluent program. Doing this optimization, however, removes the encoding of the sort hierarchy from the final \textit{CALM} program, effectively removing the capability of extending the final program manually by adding new instances to source sorts and having super sorts inherit the new instances. It is planned that future versions of \textit{CALM} will support multiple answer sets as a result of the static program, and in that case the sort hierarchy would need to be preserved.
in the final program.

Fluent Program Rules The user defined fluent axioms are normalized, translated and added to the static program to create the program $\Pi_C$ that defines the $SPARC$ encoding of the transition diagram that is the output of $CALM$ on the input system description. For dynamic causal laws with literals in the head of rules, the time variable added to these literals is $T + 1$ while the time variable added to fluent literals in the body is $T$. This step in translation models the transition between states in response to action occurrences. Fluent state constraints have $T$ added to the literal in the head to enforce the state constraint within the same time indexed state.

History Rules In the current implementation of $CALM$ the parser will accept System Descriptions which have been extended with the specification of a task and an accompanying history at the end of the $ALM$ program. The history specifies known values of a trajectory through the transition diagram. A history is composed of observations of values assigned to fluents and action occurrences at various points in time. Syntax:

\[
\text{observed}(\text{fluent}(\bar{t}, v, n)).
\]

\[
\text{happened}(<\text{action}\_\text{instance}(\bar{t}), n>).
\]

Fluent observations at time 0 specify the initial state of the trajectory. Observations at this time are translated to predicate facts in the program: Example initial state:

\[
\text{fluent}(\bar{t}, v, 0).
\]

Fluent observation at time $T > 0$ specify state constraints on the remainder of the trajectory, that it must be consistent with observations, but that the observations do not cause the trajectory to match the observed values assigned to the fluents. The
action occurrences must cause the fluent values to be assigned. Example rule based state constraint:

\[ :- \neg\text{fluent}(ar{t}, v, T). \]

Action occurrences are added as facts to the program at the time indicated. Example:

\[ \text{occurs}(\text{action}\_\text{instance}(\bar{t}), n). \]

If any observation or action occurrence is incompatible with the user defined axioms governing the definition of state and transitions for the transition diagram, no answer set will result. Currently $\text{CALM}$ does not support any facilities for debugging $\text{ALM}$ system descriptions and histories which produce no answer set. To localize the problem in the trajectory, the best approach is to start with the initial state and check if an answer set is produced. If so, incrementally extend the trajectory with time steps until the offending state transition is added. If the initial state is incompatible with the transition diagram, one knows the issue is with the definition of allowed states.

Task Execution Rules  The execution of a temporal projection task requires no additional axioms or rules apart from the translation of the history into facts of action occurrences, initial state at time step 0 and the added constraints that trajectory derived from the initial state through action occurrences must be consistent with the observed fluent states.

The execution of a planning problem requires additional axioms to model the desired goal state and to ensure that new actions are only taken after the history has ended. Let $n$ be the last time step recorded in the history. Let $m$ be the horizon at which the goal state must be reached. Let $f_1(\bar{t}_1) = v_1, \ldots, f_k(\bar{t}_k) = v_k$ be the fluent assignments in the goal state. The following rules are added to the program:

- Indicate the time steps during which the goal is achieved.
plan_goal(I) :- f₁(\(t₁, v₁, I\)),...,fₖ(\(tₖ, vₖ, I\)).

- Indicate the current time, the next time step after the last time in the history.
  current_time(n+1).

- Indicate when plan actions are allowed, at or after the current time.
  plan_allow_actions(I) :- current_time(I2),I>=I2,I<=m.

- The plan is successful when there exists a time when actions are allowed and the goal is achieved.
  plan_success :- plan_goal(I), plan_allow_actions(I).

- It is impossible to not have a successful plan.
  :- not plan_success.

- Indicate when an action occurrence is a plan action.
  plan_action(I) :- occurs(A, I), plan_allow_actions(I).

- It is impossible for a plan to skip time steps in its plan of actions.
  :- not plan_action(I), plan_action(I+1), I+1<=m, plan_allow_actions(I).

- Generate actions while planning allows actions and not goal state reached.
  occurs(A, I) :- instance(A, actions), not plan_goal(I), plan_allow_actions(I).

- It is impossible for planning to allow actions and not generation an action or be in the goal state.
:- not plan_action(I), not plan_goal(I),
     plan_allow_actions(I).

2.4.5 System Usage

The CALM executable can be obtainable from the link: https://goo.gl/NvXAZq. Download the whole folder to your computer. Examples can be found in the sub-folder examples/.

To compile an ALM system description in file f1.alm, command

java -jar calm.jar f1.alm

will output a SPARC program to standard output if there are no errors. For temporal projection in file f2.tp or planning problem in file f3.p, we have the following commands respectively:

java -jar calm.jar f2.tp
java -jar calm.jar f3.p

will output the answer sets that contain solutions to the problems.

If there is any difference between the usage above and that in the readme.txt, it is recommended to follow the instructions in readme.txt.
CHAPTER 3
ICLP

Notation In This Part

□ denotes when a value is not relevant.

Notation For Sequences

Let $A = \langle a_1, \ldots, a_n \rangle$ and $B = \langle b_1, \ldots, b_m \rangle$ be sequences.

The longest common prefix between two sequences is:
$$\text{prefix}(A, B) = \langle a_1, \ldots, a_k \rangle$$
when $a_i = b_i$ for $i \in [1..k]$ and $a_{k+1} \neq b_{k+1}$.

The length of a sequence is indicated as normal: $|A| = n$.

The symbol $\in$ denotes membership. $a_1 \in A$ and $a_1 \notin B$.

The brackets $A[i]$ denotes the $i_{th}$ element of $A$. $A[i] = a_i$.

The concatenation of two sequences is $A \triangleleft B = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$.

$\langle \rangle$ denotes the empty sequence.

Notation For Graphs and Trees

A graph $G$ is the pair $\langle V, E \rangle$ where $V$ is a set of objects, and $E$ is a subset of
$$\{ \{v_1, v_2\} | v_1 \in V \land v_2 \in V \}.$$ The elements of $V$ are called vertices or nodes. The elements of $E$ are called edges.

Given a graph $G$, $V(G)$ denotes the vertices of $G$ and $E(G)$ denotes the edges of $G$.

Given a graph $G$, a sequence of the form $\langle v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n \rangle$ is called a path from $v_0$ to $v_n$ in $G$ when for all $i \in [1..n]$, $v_i \in V(G)$, $e_i = \{v_{i-1}, v_i\} \in E(G)$ and for all $i, j \in [1..n]$, if $j \neq i$ then $e_j \neq e_i$.

Given a graph $G$, $G$ is called connected when for all $u, v \in V(G)$ where $u \neq v$, there exists a path from $u$ to $v$ in $G$. 70
Given a graph $G$, let $P = \langle v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n \rangle$ be a path in $G$, $P$ is called a cycle or cyclic path when $v_0 = v_n$.

$G = \langle \langle V, E \rangle, L_V, L_E \rangle$ is called a labeled graph when $\langle V, E \rangle$ is a graph, $L_V$ is a function from $V$ to a set of objects, and $L_E$ is a function from $E$ to a set of objects.

Given a graph $G$, if $G$ is connected and there are no cyclic paths in $G$, then $G$ is called a tree.

Given a tree $T$, for every pair of nodes $u, v \in V(T)$ where $u \neq v$, there is exactly one path from $u$ to $v$.

Proof by contradiction: Suppose there was more than one path from $u$ to $v$ in $T$. Let $\langle u = v_0, e_1, v_1, \ldots, e_n, v_n = v \rangle$ be one of the paths and $\langle u = u_0, f_1, u_1, \ldots, f_k, u_k = v \rangle$ be a different path. Since $v_0 = u_0$ and $v_n = u_k$:

- let $u_i$ and $v_i$ be such that $u_i = v_i$ and $u_{i+1} \neq v_{i+1}$,
- let $m$ and $h$ be the least integers such that, $u_m = v_h$, $m > i$, $h > i$, and for all $a \in [i+1..m-1]$ and $b \in [i+1..h-1]$, $u_a \neq v_b$.
- The path $\langle u_i, f_{i+1}, u_{i+1}, \ldots, u_{m-1}, f_m, u_m, e_h, v_{h-1}, \ldots, v_{i+1}, e_{i+1}, v_i \rangle$ is a cycle in $T$ which contradicts with $T$ being a tree.

There cannot be more than one path from any node $u$ to any node $v$ in $T$. There is at least one path since $T$ is a connected graph. Thus there is exactly one path from $u$ to $v$ in $T$ when $u \neq v$. □

Given a tree $T$, for all $v \in V(T)$, let $E_v = \{ e | e \in E(T) \land v \in e \}$, if $|E_v| = 1$ then $v$ is called a leaf of $T$.

A rooted tree is a tree with a designated vertex called the root.

Given a rooted tree $T$, for all $v \in V(T)$, let $P = \langle v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n \rangle$ be the path from the root node $v_0$ to $v_n = v$ in $T$. The node $v_{n-1}$ in $P$ is called the parent of $v$ in $T$ and $v_1, \ldots, v_{n-1}$ are called ancestors of $v$ in $T$. 

71
Given a rooted tree $T$, for all $v, u \in V(T)$, if $v$ is the parent of $u$ in $T$ then $u$ is called a \textit{child of} $v$ \textit{in} $T$, if $v$ is an ancestor of $u$ in $T$ then $u$ is called a \textit{descendant of} $v$ \textit{in} $T$.

$T = \langle \langle V, E \rangle, L_V, L_E \rangle$ is called a \textit{labeled tree} when $\langle V, E \rangle$ is a tree and $T$ is a labeled graph.

3.1 Constraint Logic Programming (CLP)

We review constraint logic programming (CLP) here.

3.1.1 CLP Program

3.1.1.1 Terms

A \textit{term} is defined recursively:

- A variable $X$ is a term.
- A string constant $t$ is a term
- The Herbrand function $f(t_1, \ldots, t_n)$ is a term when $f$ is a function symbol of arity $n$ and $t_1, \ldots, t_n$ are terms.

A term containing no variables is called a \textit{ground term}.

$s(X)$ – is an example of a term where $s/1$ is a function symbol of arity 1.

$s(s(s(1)))$ – is an example of a ground term.

3.1.1.2 Predicates

$P(t_1, \ldots, t_n)$ is called an \textit{Atom} or \textit{Predicate} of arity $n$ when $P$ is a predicate symbol and $t_1, \ldots, t_n$ are terms.

A predicate is called a \textit{ground predicate} when it contains no variables.
father(X, Y) – is an example of an atom where father/2 is a predicate symbol of arity 2.

father(abraham, isaac) – is an example of a ground predicate.

3.1.1.3 Literals

A literal l is either an atom or a primitive constraint from some constraint domain D.

X < Y * 3.14 – is a primitive constraint where X and Y are variables from \( \mathcal{R} \) and < is the less-than relation from \( \mathcal{R} \).

lessthan(X, Y * 3.14) – is a predicate representation of the above primitive constraint.

3.1.1.4 CLP Rules

A CLP rule r is of the form:

\[ l_0 \leftarrow l_1, \ldots, l_n \]

where

- \( l_0 \) is a predicate, called a user-defined constraint
- \( l_1, \ldots, l_n \) is a possibly empty sequence of arbitrary literals.

\( l_0 \) is called the head of the rule. \( l_1, \ldots, l_n \) are called the body of the rule. Rules which have a head but the body is empty are called facts and are written as \( l_0 \) without the \( \leftarrow \) symbol.

3.1.1.5 CLP Program

Let \( \Pi \) be an ordered collection of CLP rules. \( \Pi \) is called a constraint logic program (CLP Program).
For this writing we assume that $\Pi$, an arbitrary CLP program, has been given.

Let the rules in $\Pi$ be totally ordered with respect to each other. We indicate that rule $r_1$ is less than rule $r_2$ in $\Pi$ with the notation $r_1 < r_2$.

The total order on rules of $\Pi$ creates a lexicographical ordering on sequences of rules from $\Pi$. We denote this total ordering on sequences of rules from $\Pi$ as $\omega$ and use the notation $A <_\omega B$ to indicate that the sequence of rules $A$ is less than the sequence of rules $B$ with respect to $\omega$.

Note that for a set of ordered elements $L$, we refer to the least element in $L$ as the first element in $L$.

3.1.2 Queries and Derivation Trees

3.1.2.1 Query

An arbitrary sequence of literals $\langle l_1, \ldots, l_n \rangle$ from $\Pi$ is called a query.

3.1.2.2 CLP State

The pair $\langle Q | C \rangle$ is called a CLP State where $Q$ is a query and $C$ is either fail or a consistent set of primitive constraints. $C$ is called a constraint store.

The state $\langle \langle \rangle | C \rangle$ is called a success CLP state when $C$ is consistent.

In the following writing we often use $Q$ to denote the CLP state $\langle Q | \emptyset \rangle$.

3.1.2.3 Constraint Transition

Given CLP states $S$ and $S'$ and a primitive constraint $c$, $\langle S, c, S' \rangle$ is a constraint transition when

- $S = \langle Q | C \rangle$ where $C$ is a consistent set of constraints.
- $Q = \langle l_1, l_2, \ldots, l_n \rangle$ where $l_1 = c$. 
• \( Q' = \langle l_2, \ldots, l_n \rangle \).

• \( C' = C \cup \{ l_1 \} \) if it is consistent, otherwise \( C' = fail \).

• \( S' = \langle Q' | C' \rangle \).

We define \( con(\langle Q|C \rangle) \) to be \( \langle Q'|C' \rangle \) when \( \langle\langle Q|C \rangle, c, \langle Q'|C' \rangle \rangle \) is a constraint transition.

We define \( con^*(\langle Q|C \rangle) \) to be the closure of applying constraint transitions.

\[
con^*(\langle Q|C \rangle) = con(\ldots con((Q|C))\ldots).
\]

3.1.2.4 Resolution Transition

A literal \( l \) is resolvable with a rule \( r \) if the literal in the head of \( r \) has the same predicate name and arity as \( l \).

A CLP state \( \langle Q|C \rangle \) is resolvable with a rule \( r \) if the first literal in \( Q \) is resolvable with \( r \).

Given CLP states \( S \) and \( S' \) and a rule \( r \) from a program \( \Pi \), \( \langle S, r, S' \rangle \) is a resolution transition when

• \( S \) is some CLP state \( \langle Q|C \rangle \) where \( C \) is a consistent set of constraints.

• \( Q \) is of the form \( \langle l_1 = p(t_1, \ldots, t_m), \ldots, l_n \rangle \).

• \( r \) is of the form \( p(s_1, \ldots, s_m) \leftarrow B_1, \ldots, B_k \).

• \( Q' = \langle B_1, \ldots, B_k, l_2, \ldots, l_n \rangle \).

• \( C' = C \cup \{ t_1 = s_1, \ldots, t_m = s_m \} \) if it is consistent, otherwise \( C' = fail \).

• \( S' = \langle Q'|C' \rangle \).

We define \( res(\langle Q|C \rangle, r) \) to be \( \langle Q'|C' \rangle \) when \( \langle\langle Q|C \rangle, r, \langle Q'|C' \rangle \rangle \) is a resolution transition.
3.1.2.5 SLD-Derivation Tree

Given a CLP state \( \langle Q | C \rangle \) where \( C \neq \text{fail} \), an SLD-derivation tree for \( \langle Q | C \rangle \) with respect to \( \Pi \) is a labeled rooted tree \( T = \langle \langle V, E \rangle, L_V, L_E \rangle \) such that \( L_V \) is a function from \( V \) to CLP states, \( L_E \) is a function from \( E \) to rules in \( \Pi \), and \( T \) is the minimal labeled tree satisfying the following properties:

1. For the root node \( a \) in \( T \), \( L_V(a) = \text{con}^*(\langle Q | C \rangle) \).
2. For every node \( b \in V \) and for every rule \( r \in \Pi \), if \( L_V(b) \) resolves with \( r \) then there exists a node \( c \in V \) such that \( e = \{b, c\} \in E \) and \( L_V(c) = \text{con}^*(\text{res}(L_V(b), L_E(e))) \).

Given an SLD-derivation tree \( T = \langle \langle V, E \rangle, L_V, L_E \rangle \) for a CLP state \( \langle Q | C \rangle \), for every \( b \in V \), let \( P_b = \langle v_0, e_1, v_1, \ldots, e_n, v_n \rangle \) be the path from the root node \( v_0 \) to \( b = v_n \) in \( T \), the sequence of rules \( P_r = \langle L_E(e_1), \ldots, L_E(e_n) \rangle \) is called the path of derivation for \( b \) in \( T \). \( P_b \) is called the corresponding path in \( T \) for \( P_r \). If \( L_V(b) \) is of the form \( \langle \langle \rangle | C' \neq \text{fail} \rangle \) then \( P \) is called a successful path of derivation, \( C' \) is called the solution of \( P \), and the pair \( \langle C', P \rangle \) is called an annotated solution for \( \langle Q | C \rangle \).

Given a CLP state \( \langle Q | C \rangle \), the annotated solutions for \( \langle Q | C \rangle \) are totally ordered in the following way: let \( \langle C_1, P_1 \rangle \) and \( \langle C_2, P_2 \rangle \) be two different annotated solutions for \( \langle Q | C \rangle \), then \( \langle C_1, P_1 \rangle <_\omega \langle C_2, P_2 \rangle \) \textit{iff} \( P_1 <_\omega P_2 \). \footnote{We abuse notation for \( <_\omega \) to indicate when objects are ordered by components containing sequences of rules.}

Given an SLD-derivation tree \( T \) for a CLP state \( \langle Q | C \rangle \), let \( P_1, \ldots, P_n \) be all the successful paths of derivation in \( T \), in order with respect to \( \omega \). For \( i \in [1..n] \), let \( C_i \) be the solution of \( P_i \) in \( T \), \( C_i \) is called the \( i \)th solution of \( \langle Q | C \rangle \) and \( \langle C, P \rangle \) is called the \( i \)th annotated solution of \( \langle Q | C \rangle \).
3.2 The Incremental Query Problem

3.2.1 Incremental Query

We refer to a sequence of literals from $\Pi$ as a simple query to contrast it with the notion of an incremental query.

Given simple queries $Q_1, \ldots, Q_n$, the sequence $I = \langle Q_1; \ldots; Q_n \rangle$ is called an incremental query. For every $k \in [1..n]$ the prefix $I_k = \langle Q_1; \ldots; Q_k \rangle$ is also an incremental query, called a sub-incremental query of $I$. $I$ is a sub-incremental query to itself.

Given an incremental query $I = \langle Q_1; \ldots; Q_n \rangle$, let $Q_n = \langle l_{n,1}, \ldots, l_{n,k_n} \rangle$. $I$ represents the simple query $Q_1 \bowtie Q_2 \bowtie \ldots \bowtie Q_n = \langle l_{1,1}, \ldots, l_{1,k_1}, \ldots, l_{n,1}, \ldots, l_{n,k_n} \rangle$.

$\text{flatten}(I)$ denotes the simple query represented by incremental query $I$.

Given an SLD-Derivation Tree $T$ for $\text{flatten}(I)$, the $i_{th}$ solution to $\text{flatten}(I)$ is called the $i_{th}$ solution to $I$.

An annotated solution for $\text{flatten}(I)$ is called an annotated solution for $I$.

3.2.1.1 IQ Sequence

An IQ sequence is a sequence of incremental queries $\langle I_1, \ldots, I_n \rangle$ where one of the following properties holds between $I_k$ and $I_{k+1}$ for $k \in [1..n-1]$:

1. $I_i = \langle Q_1; \ldots; Q_m \rangle$, $I_{i+1} = \langle Q_1; \ldots; Q_m; Q_{m+1} \rangle$
   ($I_i$ is a sub-incremental query of $I_{i+1}$)

2. $I_i = \langle Q_1; \ldots; Q_m \rangle$, $I_{i+1} = \langle Q_1; \ldots; Q_{m-1} \rangle$
   ($I_{i+1}$ is a sub-incremental query of $I_i$)

3. $I_i = I_{i+1}$
   ($I_i$ and $I_{i+1}$ are sub-incremental queries of each other.)
3.2.1.2 IQ Commands

An *IQ command* is either *dec()*, *next()*, or *inc(Q)* where *Q* is a non-empty simple query. The command *inc(Q)* is called a *query increment* and *dec()* is called a *query decrement*.

Given a sequence of IQ commands *C* = ⟨*C*₁, ..., *C*ₙ⟩, the *IQ sequence defined by C* is
I = ⟨*I*_₀ = ⟨⟩, *I*_₁, ..., *I*_ₙ⟩ when the following properties hold:

For *i* ∈ [1..n], let *I*_ᵢ₋₁ = ⟨*Q*₁; ...; *Q*ₖ⟩:

- if *c*_ᵢ = *inc(Q)* then *I*_ᵢ = ⟨*Q*₁; ...; *Q*ₖ; *Q*⟩.
- if *c*_ᵢ = *dec()* and *k* > 1 then *I*_ᵢ = ⟨*Q*₁; ...; *Q*ₖ₋₁⟩.
- if *c*_ᵢ = *dec()* and *k* ≤ 1 then *I*_ᵢ = ⟨⟩.
- if *c*_ᵢ = *next()* then *I*_ᵢ = *I*_ᵢ₋₁.

Given a sequence of IQ commands *C* = ⟨*C*₁, ..., *C*ₙ⟩, let *I* = ⟨*I*_₀, *I*_₁, ..., *I*_ₙ⟩ be the IQ sequence defined by *C*, for each *j* ∈ [1..n] the *position of the related increment query command for I*_j* is the maximum *k* ∈ [1..j] such that *C*_ₖ = *inc(Q)* and *I*_ₖ = *I*_j*. If |*I*_j| = 0 then the position of the related increment query command is undefined. *The position of the related increment query command for C*_j* is the position of the related increment query command for *I*_j* when *C*_j* ≠ *dec()* otherwise it is the position of the related increment query command for *I*_j₋₁.

Given a sequence of IQ commands *C* = ⟨*C*₁, ..., *C*ₙ⟩, let *I* = ⟨*I*_₀, *I*_₁, ..., *I*_ₙ⟩ be the IQ sequence defined by *C*, for each *j* ∈ [1..n] where *C*_j* = *inc(Q)*, the *expiring position of C*_j* is the least integer *m* ∈ [j + 1..n] such that |*I*_m| < |*I*_j|.

Given a sequence of IQ commands *C* = ⟨*C*₁, ..., *C*ₙ⟩, let *I* = ⟨*I*_₀, ..., *I*_ₙ⟩ be the IQ sequence defined by *C*. For each *j* ∈ [1..n] where *C*_j* = *inc(Q)*, *the positions of
solution requests with respect to $C_j$ is $L = \{k|k \in [j..h], |I_k| = |I_j|\}$ where $h$ is defined as follows: If the expiring position of $C_j$ exists and it is $m$ then $h = m - 1$, otherwise $h = n$.

3.2.2 The Incremental Query Problem And Solution

Given a sequence of IQ commands $C = \langle C_1, \ldots, C_n \rangle$, the pair $\langle \Pi, C \rangle$ is called an incremental query problem (IQ problem).

Given an IQ problem $\langle \Pi, C \rangle$, let $C = \langle C_1, \ldots, C_n \rangle$ be the sequence of IQ commands, let $I = \langle I_0 = \langle \rangle, I_1, \ldots, I_n \rangle$ be the IQ sequence defined by $C$. A sequence $S = \langle S_1, \ldots, S_n \rangle$ is called an IQ Solution to $\langle \Pi, C \rangle$ when for each $i \in [1..n]$, $S_i$ satisfies the following properties:

- when $C_i = inc(Q)$, if no solution exists for $I_i$, then $S_i = fail$, otherwise $S_i$ is the first annotated solution for $I_i$.

- when $C_i = next()$, if $|I_i| = 0$ then $S_i = \Box$. If $|I_i| > 0$, then let $h$ be the position of the related increment query command for $C_i$, let $L$ be the positions of solution requests with respect to $C_h$, let $k = |\{p \in L|p \leq i\}|$. If $I_i$ has at least $k$ solutions then $S_i$ is the $k_{th}$ annotated solution for $I_i$, otherwise $S_i = fail$.

- when $C_i = dec()$ then $S_i = \Box$.

3.3 Records Of Computation

3.3.1 Computation Trees and Paths Of Computation

Given a CLP state $S$, let $R$ be the set of rules from $\Pi$ which resolve with $S$. A Choice Frame has the form $\langle S, cr, L\rangle$ such that $cr$ is either $\emptyset$ or $\{r\}$ where $r \in R$ and $L \subset R \setminus cr$. $L = \emptyset$ when $cr = \emptyset$. $L$ is called the unused resolution rules of the choice frame and $r$ is called the chosen rule of the choice frame.
Given a SLD-derivation tree $T = \langle \langle V, E \rangle, L_V, L_E \rangle$ for a CLP state $\langle Q | C \rangle$, the labeled rooted tree $T' = \langle \langle V' = V \cup \Delta_V, E' = E \cup \Delta_E \rangle, L'_V, L'_E \rangle$, where $L'_V$ is a function from $V'$ to CLP states or success and $L'_E$ is a function from $E'$ to choice frames, is called the computation tree defined by $T$ when the following properties are satisfied:

1. $|\Delta_V| = |\Delta_E|$ is the number of successful paths of derivation in $T$.

2. The root node of $T$ is the root node of $T'$.

3. For every $b \in V$ if $L_V(b)$ is a success CLP state, then there exists a node $c \in \Delta_V$ and edge $\{b, c\} \in \Delta_E$.

4. for all $b \in V'$, $L'_V(b) = L_V(b)$ when $b \in V$ otherwise $L'_V(b) = \text{success}$

5. for all $e = \{b, c\} \in E'$ if $c \in \Delta_V$ then $L'_E(e) = \langle L_V(b), \emptyset, \emptyset \rangle$

6. for all $e = \{b, c\} \in E'$ such that $b$ is the parent of $c$ and $c \not\in \Delta_V$, then $L'_E(e) = \langle L_V(b), L_E(e), L_c \rangle$ where $L_c = \{L_E(\{b, d\}) | d \text{ is a child of } b \text{ in } T, \text{ and } L_E(e) < L_E(\{b, d\}) \}$

Intuitively, if $T'$ is the computation tree defined by an SLD-derivation tree $T$, then every edge $\{b, c\}$ in $T$ is labeled by a choice frame $\langle S, cr, L \rangle$ in $T'$ where $S$ is the CLP state labeling $b$, $cr$ is the rule in $\Pi$ labeling $\{b, c\}$ in $T$, and $L$ contains all the rules labeling edges in $T$ to the children of $b$ which occur after the node $c$.

Given an SLD-derivation tree $T$ for a CLP state $\langle Q | C \rangle$, the computation tree $T'$ defined by $T$ is called a computation tree for $\langle Q | C \rangle$.

Given a computation tree $T = \langle \langle V, E \rangle, L_V, L_E \rangle$ for a CLP state $\langle Q | C \rangle$, for every node $b$ in $T$, let $\langle v_0, e_1, v_1, \ldots, e_n, v_n \rangle$ be the path in $T$ from the root node $v_0$ to $b = v_n$, the sequence of choice frames $P = \langle L_E(e_1), \ldots, L_E(e_n) \rangle$ is called the path of computation to $b$ in $T$. Let $S$ be the CLP state of $L_E(e_n)$. The path of computation
\[ \langle L_E(e_1), \ldots, L_E(e_{n-1}) \rangle \] is called a path of computation to \( S \) in \( T \). If \( S \) is a success CLP state then \( P \) is called a successful path of computation for \( \langle Q|C \rangle \).

Given the computation tree \( T' \) defined by an SLD-derivation tree \( T \) for a CLP state \( \langle Q|C \rangle \), let \( P = \langle F_1, \ldots, F_n \rangle \) be the path of computation to a node \( b \) in \( T' \). If \( b \) is in \( T \) then \( \text{rules}(P) \) denotes the path of derivation to \( b \) in \( T \). If \( b \) is not in \( T \) then let \( c \) be the parent node of \( b \) in \( T' \), \( c \) is in \( T \) and \( \text{rules}(P) \) denotes the path of derivation to \( c \) in \( T \). Note that the sequence of chosen rules in the choice frames of \( P \) correspond to the sequence of rules in \( \text{rules}(P) \).

Given a computation tree \( T \) for some CLP state \( \langle Q|C \rangle \), let \( P \) be a successful path of computation in \( T \). The annotated solution derived from \( P \) is \( \langle K, \text{rules}(P) \rangle \) where \( K \) is the constraint store of the last choice frame on \( P \).

Given a computation tree \( T \) for some CLP state \( \langle Q|C \rangle \) and a simple query \( Q' \), let \( P \) be a path of computation in \( T \). The expansion of \( P \) by \( Q' \) is the sequence of choice frames \( \langle F'_1, \ldots, F'_n \rangle \) where for all \( i \in [1..n] \), \( F_i = \langle \langle Q_i|C_i \rangle, cr_i, L_i \rangle \) and \( F'_i = \langle \langle Q_i \leftarrow Q|C_i \rangle, cr_i, L_i \rangle \). \( \text{exp}(P, Q') \) denotes the expansion of \( P \) by \( Q' \).

Note that if \( P \) is a path of computation in a computation tree for some CLP state \( \langle Q|C \rangle \) then

- \( \text{exp}(P, Q') \) is a path of computation in a computation tree for CLP state \( \langle Q \leftarrow Q'|C \rangle \).
- \( \text{exp}(\text{exp}(P, Q_1), Q_2) = \text{ext}(P, Q_1 \leftarrow Q_2) \).
- \( \text{rules}(\text{exp}(P, Q)) = \text{rules}(P) \).

Given a simple query \( Q \) and paths of computation \( P, P' \), and \( P'' \) such that \( P = \text{exp}(P', Q) \leftarrow P'' \), \( P \) is called an extension of \( P' \) with respect to \( Q \).
Given an incremental query $I = \langle Q_1; \ldots; Q_n \rangle$ and a successful path of computation $P$ for $\text{flatten}(I)$, let $I' = \langle Q_1; \ldots; Q_m \rangle$ be a sub-incremental query to $I$. A successful path of computation $P'$ for $\text{flatten}(I')$ is called the \textit{precedent path of $P$ for $I'$} when $P$ is an extension of $P'$ with respect to $Q_{m+1} \leadsto \ldots \leadsto Q_n$. $\text{precedent}(P, I')$ denotes the precedent path of $P$ for $I'$.

Note that $\text{rules}(\text{precedent}(P, I'))$ is a prefix of $\text{rules}(P)$.

Note that $\text{precedent}(\text{precedent}(P, I'), I'') = \text{precedent}(P, I'')$ when there exists an incremental query $I$ such that $I'$ is a sub-incremental query of $I$, $I''$ is a sub-incremental query of both $I$ and $I'$, and $P$ is a path of computation for $\text{flatten}(I)$.

Given a computation tree $T'$ defined by an SLD-derivation tree $T$ for a CLP state $\langle Q|C \rangle$, let $P_1, \ldots, P_n$ be the paths of computation to all leaf nodes in $T'$. The sequence $\langle P_1, \ldots, P_n \rangle$ is a \textit{representation of $T'$} when for all $i, j \in [1..n]$ such that $i < j$, $\text{rules}(P_i) <_\omega \text{rules}(P_j)$.

From this point forward we represent computation trees as sequences of paths of computation.

Given a computation tree $T$ for $\langle Q|C \rangle$, the maximum subsequence of successful paths of computation in $T$ is called a \textit{complete success tree for $\langle Q|C \rangle$}.

Given a complete success tree $T$ for $\langle Q|C \rangle$, any prefix of $T$ is called a \textit{partial success tree for $\langle Q|C \rangle$}. The empty sequence $\langle \rangle$ is a trivial partial success tree for $\langle Q|C \rangle$.

### 3.3.2 Record Of Computation

Given an incremental query $I = \langle Q_1; \ldots; Q_n \rangle$, for $i \in [1..n]$, let $I_i = \langle Q_1; \ldots; Q_i \rangle$ and let $T_i$ be a partial success tree for $\text{flatten}(I_i)$. The sequence $R = \langle T_1, \ldots, T_n \rangle$ is called a \textit{record of computation for $I$} when the following properties hold:

1. For $i \in [2..n]$, for every $P \in T_i$, $\text{precedent}(P, I_{i-1}) \in T_{i-1}$
2. Every path of computation in $T_1, \ldots, T_n$ is marked as *closed* or *open* under the following restrictions:

- all paths in $T_n$ are marked as *open*.
- for $i \in [1..n - 1]$, for all paths $P_i \in T_i$, if $P_i$ is marked as closed then $T_{i+1}$ contains all the successful paths of computation for $\text{flatten}(I_{i+1})$ which are extensions of $P_i$.

### 3.4 IQ Transition Diagram

We now describe a state transition diagram of sound paths of computation which a solver may take for computing IQ solutions to arbitrary IQ problems. We call the transition diagram the *Incremental Query Transition Diagram for program $\Pi$*, denoted as $IQTD(\Pi)$.

The state description of the solver includes the structure of the current incremental query, a record of computation containing annotated solutions to the incremental query and its sub-incremental queries, the current CLP state and path of exploration, and what mode the search procedure is in.

#### 3.4.1 IQ State Of Computation

An *IQ state* is of the form:

$$\langle \langle I_A, I_D \rangle, R, N, P, \langle Q|C \rangle, M \rangle$$

where:

$I_A$ and $I_D$ are sequences of simple queries.

$I_A \hookrightarrow I_D = \langle I_1, \ldots, I_n \rangle$ is called the *incremental query of the IQ state*, $I_A$ is called the *active sub-incremental query*, and $I_D$ is called the *dormant queries of $I_A \hookrightarrow I_D$*. 

83
$R = \langle T_1, \ldots, T_n \rangle$ is a record of computation for $I_A \leadsto I_D$.

$N = \langle N_1, \ldots, N_n \rangle$ is a sequence of integers with the following properties:

- $1 \leq N_i \leq |T_i|$ for $i \in [1..n - 1]$
- $1 \leq N_n \leq |T_n| + 1$

A record of computation may contain more solutions than what have been requested through IQ commands. $N$ keeps track of how many solutions have been requested by IQ commands for each sub-incremental query of the current query.

$\langle Q|C \rangle$ is either a CLP state or $\langle \Box|\Box \rangle$.

$P$ is either a path of computation or $\Box$.

$M$ is either proceed, backtrack, or halt.

- when $M = halt$ then $P = \Box$, $\langle Q|C \rangle = \langle \Box|\Box \rangle$.
- when $M = backtrack$ then $P \neq \Box$, $\langle Q|C \rangle = \langle \Box|\Box \rangle$.
- when $M = proceed$ then $P \neq \Box$, $\langle Q|C \rangle \neq \langle \Box|\Box \rangle$.

When $\langle I_A, I_D \rangle = \langle \langle Q_1; \ldots; Q_k, Q_{k+1}; \ldots; Q_n \rangle \rangle$

Let $C_{k-1}$ be the solution of the first open path in $T_{k-1}$.

- when $M$ is proceed or backtrack and $|P| > 0$ then the CLP state of the first choice frame in $P$ is $\langle Q_k|C_{k-1} \rangle$
- when $M$ is proceed and $P = \langle \rangle$ then $\langle Q|C \rangle = \langle Q_k|C_{k-1} \rangle$
- when $M$ is proceed and $|P| > 0$, let $\langle (Q_P|C_P), \{r\}, L \rangle$ be the last choice frame of $P$, then $\langle Q|C \rangle$ the result of $\text{con}^k(\text{res}(\langle Q_P|C_P \rangle, r))$ where $\text{con}^k$ is 0 or more applications of constraint resolution.
The initial IQ state is of the form:

\[ \langle \langle I_A = \langle \rangle, I_D = \langle \rangle, N = \langle \rangle, R = \langle \rangle, P = \Box, \langle \Box \Box \rangle, \text{halt} \rangle \]

A success IQ state is of the form:

\[ \langle \langle I_A, I_D = \langle \rangle, R = \langle T_1, \ldots, T_n \rangle, N = \langle N_1, \ldots, N_n \rangle, \Box, \langle \Box \Box \rangle, \text{halt} \rangle \]

where \( |T_n| = N_n \)

A fail IQ state is of the form:

\[ \langle \langle I_A, I_D = \langle \rangle, R = \langle T_1, \ldots, T_n \rangle, N = \langle N_1, \ldots, N_n \rangle, \Box, \langle \Box \Box \rangle, \text{halt} \rangle \]

where \( |T_n| < N_n \)

The initial, success and fail IQ states are called halted states and are the only IQ states where \( M = \text{halt} \).

3.4.2 IQ State Transitions

The state transitions are organized around what mode the search procedure is in. From a halted state, the solver can receive IQ commands which modify the incremental query and record of computation. Searching for the next solution to an incremental query \( \langle Q_1; \ldots; Q_n \rangle \) is done in the context of saved solutions to the immediate sub-incremental query \( \langle Q_1; \ldots; Q_{n-1} \rangle \). If all saved solutions have been exhausted, it becomes necessary to search for new solutions to the sub-incremental query. State transitions model the book keeping about which sub-incremental query is being answered and which solutions have been used in the record of computation. Backtracking must be explicitly modeled to account for re-use of the information.
saved in the record of computation.

An IQ state transition has the form

$$\langle S, e, S' \rangle$$

where

- $S$ and $S'$ are IQ states
- $e \in \{inc(Q), dec(), nexp(), c, r, save, backtrack, fail\}$ where $Q$ is a simple query, $c$ is a primitive constraint, and $r$ is either $\square$ or a rule from $\Pi$.

Transitions are divided into 3 categories:

- **command transitions** - process IQ commands
- **proceed transitions** - model “forward” computation searching an SLD-Derivation tree
- **backtrack transitions** - carry saved information “backwards” during backtrack operations.

3.4.2.1 Command Transitions

**Increment Query Command Transition**

Let $inc(Q)$ be an IQ command. $\langle S, inc(Q), S' \rangle$ is a command transition when:

$$S = \langle \langle I_A, I_D = \langle \rangle \rangle, R, N, \square, \langle \square|\square \rangle, halt \rangle$$

$$S' = \langle \langle I'_A, I'_D = \langle \rangle \rangle, R', N', P', \langle Q'|C' \rangle, M' \rangle$$

where
• \( I_A = \langle Q_1; \ldots; Q_n \rangle \)

• \( I'_A = \langle Q_1; \ldots; Q_n; Q \rangle \)

• \( R = \langle T_1, \ldots, T_n \rangle \)

• \( R' = \langle T_1, \ldots, T_n, \rangle \rangle \)

• \( N = \langle N_1, \ldots, N_n \rangle \)

• \( N' = \langle N_1, \ldots, N_n, 1 \rangle \)

• If \( S \) is the initial IQ state then \( P' = \langle \rangle, Q' = Q, C' = \emptyset \), and \( M' = \text{proceed} \).

• If \(|T_n| = 0\) then \( P' = \Box, \langle Q'|C' \rangle = \langle \Box|\Box \rangle \) and \( M' = \text{halt} \).

• Otherwise

  – Every path in \( T_n \) is marked as open.

  – \( P_n \) is the first path in \( T_n \),

  – \( P' = \langle \rangle, Q' = Q, C' \) is the solution of \( P_n \) and \( M' = \text{proceed} \).

**Next Solution Command Transition**

If \( S \) is the initial IQ state, then \( \langle S, \text{next}(), S \rangle \) is a command transition.

\( \langle S, \text{next}(), S' \rangle \) is a command transition when:

\[
S = \langle \langle I_A, I_D = \langle \rangle \rangle, R, N, \Box, \langle \Box|\Box \rangle, \text{halt} \rangle \\
S' = \langle \langle I_A, I_D = \langle \rangle \rangle, R, N', P', \langle \Box|\Box \rangle, M' \rangle
\]

where

• \( I_A = \langle Q_1; \ldots; Q_n \rangle \)
\[ R = \langle T_1, \ldots, T_n \rangle \]
\[ N = \langle N_1, \ldots, N_n \rangle \]
\[ N' = \langle N_1, \ldots, N_{n-1}, N_n + 1 \rangle \]

- If \(|T_n| \neq N_n\) then \(P' = \Box\) and \(M' = halt\)

- If \(|T_n| = N_n\) then let \(P_n\) be the last path in \(T_n\), \(P'\) is such that \(P_n = exp(P'', Q_n) \leftarrow P'\) for some \(P'' \in T_{n-1}\) and \(M' = backtrack\).

**Decrement Query Command Transition**

If \(S\) is the initial IQ state, then \(\langle S, dec() , S \rangle\) is a command transition.

\(\langle S, dec() , S' \rangle\) is a command transition when:

\[
S = \langle \langle I_A, I_D = \langle \rangle \rangle, R, N, \Box, \langle \Box | \Box \rangle, halt \rangle \\
S' = \langle \langle I'_A, I_D = \langle \rangle \rangle, R', N', \Box, \langle \Box | \Box \rangle, halt \rangle
\]

where

- \(I_A = \langle Q_1; \ldots; Q_n \rangle\) (\(n > 0\))

- \(I'_A = \langle Q_1; \ldots; Q_{n-1} \rangle\)

- \(R = \langle T_1, \ldots, T_n \rangle\)

- \(R' = \langle T_1, \ldots, T_{n-1} \rangle\)

- \(N = \langle N_1, \ldots, N_n \rangle\)

- \(N' = \langle N_1, \ldots, N_{n-1} \rangle\)

- all paths in \(T_{n-1}\) are marked as open.
3.4.2.2 Proceed Transitions

Proceed transitions attempt to make progress towards finding the $N[n]_{th}$ solution of $I_A \sim I_D$. The Constraint Transition and Resolution Transition model exploring a particular path of derivation to a new solution for $I_A$. The Intermediate Save Transition records the discovery of new solutions to $I_A$, when $I_D \neq \langle \rangle$. The Final Save Transition halts searching and records the $N[n]_{th}$ solution to $I_A$ when $I_D = \langle \rangle$. If a Constraint Transition or Resolution Transition cannot make forward progress, then they will set the necessary state to initiate backtracking within the current active sub-incremental query.

Constraint Transition

Let $c$ be a primitive constraint. $\langle S, c, S' \rangle$ is a proceed transition when:

$$S = \langle \langle I_A, I_D \rangle, R, N, P, \langle Q|C \rangle, proceed \rangle$$
$$S' = \langle \langle I_A, I_D \rangle, R, N, P, \langle Q'|C' \rangle, M' \rangle$$

where

- $Q$ has the form $\langle c, l_2, \ldots, l_m \rangle$
- Let $\langle Q_c|C_c \rangle = con(\langle Q|C \rangle)$
- if $C_c = fail$ then $\langle Q'|C' \rangle = \langle \square|\square \rangle$ and $M' = backtrack$
- Otherwise $\langle Q'|C' \rangle = \langle Q_c|C_c \rangle$ and $M = proceed$.

Resolution Transition

Let $r$ be a rule from $\Pi$ or $\square$. $\langle S, r, S' \rangle$ is a proceed transition when:

$$S = \langle \langle I_A, I_D \rangle, R, N, P, \langle Q|C \rangle, proceed \rangle$$
$$S' = \langle \langle I_A, I_D \rangle, R, N, P', \langle Q'|C' \rangle, M' \rangle$$
where

- The first literal in $Q$ is a user defined constraint.
- $L$ is the set of all rules from $\Pi$ that resolves with $\langle Q|C \rangle$.
- If $L = \emptyset$ then $r = \Box$, $P' = P$, $\langle Q'|C' \rangle = \langle \Box|\Box \rangle$, and $M' = backtrack$.
- Otherwise
  - Let $r$ be the first rule in $L$.
  - $P' = P \smallsetminus \langle \langle \langle Q|C \rangle, \{r\}, L \setminus \{r\} \rangle \rangle$
  - Let $\langle Q_r|C_r \rangle = res(\langle Q|C \rangle, r)$
  - if $C_r = fail$ then $\langle Q'|C' \rangle = \langle \Box|\Box \rangle$ and $M' = backtrack$
  - otherwise $\langle Q'|C' \rangle = \langle Q_r|C_r \rangle$ and $M = proceed$

**Intermediate Save Transition**

$\langle S, save, S' \rangle$ is a proceed transition when:

$$S = \langle \langle I_A, I_D \rangle, R, N, P, \langle \{}|C \rangle, proceed \rangle$$

$$S' = \langle \langle I'_A, I'_D \rangle, R', N, \langle \}, \langle Q'|C \rangle, proceed \rangle$$

where

- $I_A = \langle Q_1; \ldots; Q_k \rangle$ and $I_D = \langle Q_{k+1}; \ldots; Q_n \rangle$
- $I'_A = \langle Q_1; \ldots; Q_k; Q_{k+1} \rangle$ and $I'_D = \langle Q_{k+2}; \ldots; Q_n \rangle$
- $R = \langle T_1, \ldots, T_n \rangle$
- $P_{k-1}$ is the first open path in $T_{k-1}$.
- $P_k = exp(P_{k-1}, Q_k) \smallsetminus P \smallsetminus \langle \langle \emptyset|C \rangle, \emptyset, \emptyset \rangle$
\[ R' = (T_1, \ldots, T_{k-1}, T_k \sim \langle P_k \rangle, T_{k+1}, \ldots, T_n) \]

- \( P_k \) is marked as open in \( R' \).
- \( Q' = Q_{k+1} \)

**Final Save Transition**

\( \langle S, save, S' \rangle \) is a proceed transition when:

\[
S = \langle \langle I_A, I_D = \langle \rangle \rangle, R, N, P, \langle \langle \langle C \rangle \rangle, proceed \rangle \rangle \\
S' = \langle \langle I_A, I_D = \langle \rangle \rangle, R', N, \Box, \langle \Box \Box \rangle, halt \rangle \\
\]

where

- \( I_A = \langle Q_1; \ldots; Q_n \rangle \).
- \( R = \langle T_1, \ldots, T_n \rangle \).
- \( R' = \langle T_1, \ldots, T_{n-1}, T_n \sim \langle P_n \rangle \rangle \) where \( P_{n-1} \) is the first open path in \( T_{n-1} \) and \( P_n = exp(P_{n-1}, Q_n) \sim P \sim \langle \langle C \rangle \rangle, \emptyset, \emptyset \rangle \).
- \( P_n \) is marked as open in \( T_n \).

**3.4.2.3 Backtrack Transitions**

The traditional search of an SLD-Derivation tree does not formally model backtracking. When a state is returned to through backtracking, it is assumed that the information generated by exploring the sub-tree is no longer relevant. In our case we are saving any new solutions to sub-incremental queries we’ve encountered in the exploration of a failed sub-tree and we must formally model carrying this information back up the paths of derivation. Since we are searching for solutions of incremental queries in the context of solutions to their sub-incremental queries, backtracking must behave differently in the following cases:
• backtracking within \( Q_m | C_{m-1} \) when \( I_A = \langle Q_1; \ldots; Q_m \rangle \) and \( C_{m-1} \) is a solution for \( \langle Q_1; \ldots; Q_{m-1} \rangle \).

• backtracking to the next saved solution \( C'_{m-1} \) for \( \langle Q_1; \ldots; Q_{m-1} \rangle \) to continue searching for a new solution to \( I_A \). (The subtree below \( Q_m | C_{m-1} \) has been exhausted and no new solution was found).

• backtracking to find a new solution to \( \langle Q_1; \ldots; Q_{m-1} \rangle \) when there are no more saved solutions \( C'_{m-1} \) to search under.

• Entering a fail state when no new solution could be found to \( I_A \bowtie I_D \).

\textbf{Backtrack Within Current Query Transition}

\( \langle S, \text{backtrack}, S' \rangle \) is a backtrack transition when:

\[
S = \langle \langle I_A, I_D \rangle, R, N, P \neq \langle \rangle, \langle \square|\square \rangle, \text{backtrack} \rangle
\]

\[
S' = \langle \langle I_A, I_D \rangle, R, N, P', \langle Q'|C' \rangle, M' \rangle
\]

where

• \( P = \langle F_1, \ldots, F_m \rangle \).

• \( F_i = \langle \langle Q_{F_i}|C_{F_i} \rangle, r_{F_i}, L_{F_i} \rangle \) is the last choice frame on \( P \) such that \( L_{F_i} \neq \emptyset \).

• Let \( r \) be the first rule in \( L_{F_i} \).

• \( P' = \langle F_1, \ldots, F_{i-1} \bowtie \langle \langle Q_{F_i}|C_{F_i} \rangle, \{r\}, L_{F_i} \setminus \{r\} \rangle \).

• \( \langle Q_r|C_r \rangle = \text{res}(\langle Q_{F_i}|C_{F_i} \rangle, r) \)

• if \( C_r = \text{fail} \) then \( \langle Q'|C' \rangle = \langle \square|\square \rangle \) and \( M' = \text{backtrack} \).

• otherwise \( \langle Q'|C' \rangle = \langle Q_r|C_r \rangle \) and \( M = \text{proceed} \).
Next Existing Solution To Previous Query Transition

\[ \langle S, \text{backtrack}, S' \rangle \] is a backtrack transition when:

\[
S = \langle \langle I_A, I_D \rangle, R, N, P, \langle \Box|\Box \rangle, \text{backtrack} \rangle \\
S' = \langle \langle I_A, I_D \rangle, R', N, \langle \Box \rangle, \langle Q'|C' \rangle, \text{proceed} \rangle
\]

where

- \( P \) has no choice frame with unused resolution rules.
- \( I_A = \langle Q_1; \ldots; Q_k \rangle \) and \( I_D = \langle Q_{k+1}; \ldots; Q_n \rangle \) with \( k \geq 1 \)
- \( R = \langle T_1, \ldots, T_n \rangle \)
- There are 2 or more open paths in \( T_{k-1} \)
- Let \( P_1 \) and \( P_2 \) be the first and second paths marked as open in \( T_{k-1} \).
- \( R' \) is \( R \) with \( P_1 \) marked as closed.
- \( Q' = Q_k \) and \( C' \) is the solution of \( P_2 \).

Find New Solution To Previous Query Transition

\[ \langle S, \text{backtrack}, S' \rangle \] is a backtrack transition when:

\[
S = \langle \langle I_A, I_D \rangle, R, N, P, \langle \Box|\Box \rangle, \text{backtrack} \rangle \\
S' = \langle \langle I_A', I_D' \rangle, R', N, P', \langle \Box|\Box \rangle, \text{backtrack} \rangle
\]

where

- \( P \) has no choice frame with unused resolution rules.
- \( I_A = \langle Q_1; \ldots; Q_k \rangle \) and \( I_D = \langle Q_{k+1}; \ldots; Q_n \rangle \) with \( k \geq 2 \)
- \( I'_A = \langle Q_1; \ldots; Q_{k-1} \rangle \) and \( I'_D = \langle Q_k; Q_{k+1}; \ldots; Q_n \rangle \)

- \( R = \langle T_1, \ldots, T_n \rangle \)

- There is only one open path in \( T_{k-1} \), and let \( P_{k-1} \) be that path.

- \( P_{k-2} = \text{precedent}(P_{k-1}, \langle Q_1, \ldots, Q_{k-2} \rangle) \)

- \( P' \) is such that \( P_{k-1} = \exp(P_{k-2}, Q_{k-1}) \leadsto P' \)

- \( R' \) is \( R \) with \( P_{k-1} \) marked as closed.

**Fail Transition**

\( \langle S, \text{fail}, S' \rangle \) is a backtrack transition when:

\[
S = \langle \langle I_A, I_D \rangle, R, N, P, \langle \Box|\Box \rangle, \text{backtrack} \rangle
\]

\[
S' = \langle \langle I'_A, I'_D = \langle \rangle \rangle, R, N, \Box, \langle \Box|\Box \rangle, \text{halt} \rangle
\]

where

- \( P \) has no choice frame with unused resolution rules.

- either \( k = 1 \), or when \( k > 2 \) there is only 1 open path in \( T_{k-1} \) and no open paths in \( T_{k-2} \).

- \( I'_A = I_A \leadsto I_D \).

### 3.4.3 IQTD Path

Given a CLP program \( \Pi \), we use the term \( IQTD(\Pi) \) to denote the state diagram defined by the IQ state transitions.

Given the state diagram \( IQTD(\Pi) \), let \( P = \langle s_0, e_1, s_1, \ldots, e_n, s_n \rangle \) be a sequence of labels on states and edges in \( IQTD(\Pi) \) such that for each \( i \in [1..n], \langle s_{i-1}, e_i, s_i \rangle \) is an
IQ state transition, $P$ is called a path in $IQTD(\Pi)$. If $s_0$ is the initial IQ state, then $P$ is called a sound path. If $s_n$ is a relevant IQ state, and $P$ is a sound path, then $P$ is called a halted sound path.

Given a CLP program $\Pi$ and a halted path $P = \langle s_0, e_1, s_1, \ldots, e_n, s_n \rangle$ from $IQTD(\Pi)$, let $C = \langle C_1, \ldots, C_k \rangle$ be the maximal subsequence of IQ commands labeling edges along $P$, $C$ is called the sequence of IQ commands defined by $P$.

Given a CLP program $\Pi$, let $P$ be a halted sound path in $IQTD(\Pi)$, let $C = \langle C_1, \ldots, C_n \rangle$ be the sequence of IQ commands defined by $P$ and let $L = \langle L_0, \ldots, L_n \rangle$ be the sequence of halted IQ states along $P$. For $i \in [0..n]$:

- let $L_i = \langle \langle I_{A_i}, \langle \rangle \rangle, R_i, N_i, \square, \langle \square | \square \rangle, halt \rangle.$
- let $R_i = \langle T_{i,1}, \ldots, T_{i,k_i} \rangle.$
- let $N_i = \langle N_{i,1}, \ldots, N_{i,k_i} \rangle.$
- let $T_{i,k_i} = \langle P_{i,1}, \ldots, P_{i,m_i} \rangle.$

The IQ solution defined by $P$ is the sequence $S = \langle S_1, \ldots, S_n \rangle$ such that for $j \in [1..n]$:

- If $C_j = dec()$, then $S_j = \square$
- If $C_j = next()$ and $I_{A_j} = \langle \rangle$, then $S_j = \square$
- If $C_j \neq dec()$, $I_{A_j} \neq \langle \rangle$ and $|T_{j,k_j}| < N_{j,k_j}$, then $S_j = fail$.
- Otherwise let $h = N_{j,k_j}$, $S_j$ is the annotated solution derived from $P_{j,h}$.

**Theorem** Given an IQ problem $\langle \Pi, C = \langle C_1, \ldots, C_n \rangle \rangle$, $S = \langle S_1, \ldots, S_n \rangle$ is an IQ solution to $\langle \Pi, C \rangle$ if and only if there exists a halted sound path $P$ in $IQTD(\Pi)$ such
that $C$ is the sequence of IQ commands defined by $P$ and $S$ is the IQ solution defined by $P$.

The proof is found in the appendix C.4.
CHAPTER 4
CONCLUSION AND FUTURE WORK

4.1 \textit{CALM} Conclusion

\textit{CALM} is the first robust compiler for \textit{ALM} supporting the full syntax and able to handle malformed system descriptions through providing semantic errors. \textit{CALM} has already been used in the context of \textit{Processing Narratives by Means of Action Languages}[39]. Without a robust compiler it is rare for a new programming language to be investigated for its effectiveness as a modeling language. We hope the existence of \textit{CALM} will allow enable the \textit{ALM} approach to action languages and modeling dynamic domains.

In the original \textit{ALM} paper[24] it was suggested that an instance definition \( D \) for a non-source sort \( S \) with sub sorts could be implemented by adding a disjunctive clause which would investigate \( D \) as being \( is\_a \) of each source sub-sort of \( S \). Due to limitations with \textit{SPARC}'s implementation of sorts, this is not possible. \textit{SPARC} requires that the sort declarations be total and closed. There is no facility for the hypothetical consideration of elements in sub-sorts in the hierarchy.

The original paper did not explain the concept of a parameterized constant as used in the monkey banana problem. Implementation of constant definitions in the structure proved complex for non-ground constants. Our implementation only supports the definition of ground constants. Parameterized constant declarations in the theory are supported. Instances are treated as belonging to an anonymous subsort of the declared sort for the constants.

4.2 \textit{CALM} Future Work

There are many basic extensions to the \textit{ALM} syntax that would expand utility and expressiveness of the language. The following extensions could be implemented
with the $SPARC$ solver:

- Adding support for aggregate literals[41, 8, 13] in the body of axioms.
- Adding support for discrete time parameters in fluent axioms such as dynamic causal laws. This would allow the specification of the effects of actions to occur later than the next time step.
- Replacing the implementation of parameterized constants with support for parameterized sort declarations whose instances are automatically instantiated through schema definitions.
- Adding support for additional constraints on generated plans in planning problems.
- Adding support for diagnosis and explanation problems[9].

In order to expand capability of $ALM$ to continuous constraint domains, translation to a constraint enabled $ASP$ solver is required. One such solver is $AC(C)$.

4.3 $ICLP$ Conclusion

Our work in the development of an Incremental Query Transition Diagram ($IQTD$) extends and generalized the work of Peter Stuckey[31] and Pascal Van Hentenryck[43]. Our transition diagram models the state of an sld-resolution based solver that is capable of saving the solution and path of computation for each sub-incremental query in the query stack when they are discovered. This transition diagram will form the basis for modifying the $CLAM(\mathcal{R})$ solver with new instructions for saving incremental solutions. An Incremental $CLAM(\mathcal{R})$ solver will allow for an implementation of the $ACSolver$ which does not have to restart the $CLP$ solver each time a query is withdrawn during backtracking on guesses.
4.4 ICLP Future Work

The next step towards the development of an AC Solver which utilizes an incremental CLAM(R) solver is to optimize the IQTD states to eliminate redundant representation of common paths to solutions between nested sld-derivation trees. The successful paths of computation saved at level $n$ should reference and extend successful paths of computation saved for level $n - 1$. After this optimization, the successful paths of computation reflect the paths of computation saved in the WAM and CLAM(R). The semantics of the existing instruction set can be modified and augmented to facilitate saving and restoring both paths of computation and solutions in the constraint store.


101


APPENDICES

Appendix A

ALM Grammar in ANTLR4

/**
 * This Grammar Follows the BNF for ALM as described in
 * Appendix A of "Modular Action Language ALM"
 * by Daniela Inclezan and Michael Gelfond.
 *
 * This document was created by Edward Wertz
 * Date: 7/2/2015
 * Copyright: Texas Tech University
 */

grammar ALM;

/*@ ABOUT THE GRAMMAR RULES THAT FOLLOW:
* 1) Lexer Rules and TOKEN names start with capitol letters.
* 2) parser rules start with lowercase letters
*
* Parser rules are clustered based on their "rank", "rank" is
* the maximum distance from lexer tokens in the BNF grammer
* The top-level rule will be the last rule in the grammar.
*/

/*@ LEXER RULES
*/
// ORDER OF RULES IN FILE (TOP TO BOTTOM) matters
// ORDER OF DEFINITIONS (LEFT TO RIGHT) within
// NON-TERMINAL and TOKENS matters

COMMENT:   '//' '[^r\n]* ('r' '?' '\n' | EOF) -> skip;

WhiteSpace: (' ' '|' '\t' '|' '\r' '|' '\n') -> skip; //SKIP WHITESPACE

//EAGERLY CREATE THESE SPECIFIC TOKENS
MOD: 'mod';
EQ: '=';// Describes <eq>
NEQ: '!='; //Describes <neq>
ARITH_OP: '+' | '-' | '*' | '/' | 'mod' | '^'; //<arithmetic_op>
COMP_REL: '>' | '<' | '<=' | '>='; // <comparison_rel>
RIGHT_ARROW: '->'; //Used in describing Function signatures.
OCCURS: 'occurs'; //key word
INSTANCE: 'instance'; //key word
IS_A: 'is_a'; //key word
HAS_CHILD: 'has_child'; //special function
HAS_PARENT: 'has_parent'; //special function
LINK: 'link'; //special function
SOURCE: 'source'; //special function
SINK: 'sink'; //special function
SUBSORT: 'subsort'; //special function
DOM: 'DOM'; // key word
SORT: 'sort';
STATE: 'state';
CONSTRAINTS: 'constraints';
FUNCTION: 'function';
DECLARATIONS: 'declarations';
DEFINITIONS: 'definitions';
SYSTEM: 'system';
DESCRIPTION: 'description';
THEORY: 'theory';
MODULE: 'module';
IMPORT: 'import';
FROM: 'from';
DEPENDS: 'depends';
ON: 'on';
ATTRIBUTES: 'attributes';
OBJECT: 'object';
CONSTANT: 'constant';
STATICS: 'statics';
FLUENTS: 'fluents';
BASIC: 'basic';
DEFINED: 'defined';
TOTAL: 'total';
AXIOMS: 'axioms';
DYNAMIC: 'dynamic';
CAUSAL: 'causal';
LAWS: 'laws';
EXECUTABILITY: 'executability';
CONDITIONS: 'conditions';
CAUSES: 'causes';
IMPOSSIBLE: 'impossible';
IF: 'if';
FALSE: 'false';
TRUE: 'true';
STRUCTURE: 'structure';
IN: 'in';
WHERE: 'where';
VALUE: 'value';
OF: 'of';
INSTANCES: 'instances';
TEMPORAL: 'temporal';
PROJECTION: 'projection';
MAX: 'max';
STEPS: 'steps';
HISTORY: 'history';
OBSERVED: 'observed';
HAPPENED: 'happened';
PLANNING: 'planning';
PROBLEM: 'problem';
DIAGNOSTIC: 'diagnostic';
GOAL: 'goal';
SITUATION: 'situation';
WHEN: 'when';
NORMAL: 'normal';
ACTION: 'action';
ADDITIONAL: 'additional';
RESTRICTIONS : 'restrictions';
PERMISSIONS : 'permissions';
POSSIBLE: 'possible';
AVOID: 'avoid';
BOOLEANS: 'booleans';
INTEGERS: 'integers';
UNIVERSE: 'universe';
ACTIONS: 'actions';
CURRENT: 'current';
TIME: 'time';

// THESE TOKENS ARE MORE GENERAL AND LESS EAGERLY DETERMINED
ID: [a-z]([a-zA-Z0-9_\-]*)*; //<identifier>
VAR: [A-Z]([a-zA-Z0-9_\-]*)*; //<variable>
POSINT: [1-9][0-9]*; //<positive_integer>
NEGINT: '-'[1-9][0-9]*; //<negative_integer>
ZERO: [0]+; //ZERO

/*
 * ALM BNF RULES subsumed by LEXER rules
 * -------------------------------
 * <boolean> -> BOOL
 * <non_zero_digit> -> POSINT and NEGINT
 * <digit> -> POSINT and NEGINT and ID and VAR
 * <lowercase_letter> -> ID and VAR
 * <uppercase_letter> -> ID and VAR
 */
* <letter> -> ID and VAR
* <identifier> -> ID
* <variable> -> VAR
* <positive_integer> -> POSINT
* <integer> -> ZERO and POSINT and NEGINT
* <arithmetic_op> -> ARITH_OP
* <comparison_rel> -> COMP_REL
*/

/*
* BASIC PARSER RULES BUILT OUT OF SPECIAL LEXER TOKENS
*/

bool : TRUE | FALSE;

nat_num : ZERO | POSINT; /*natural_number*/
integer : ZERO | POSINT | NEGINT; /*integer*/
relation: EQ | NEQ | COMP_REL; /*arithmetic_rel*/

// RECOVER KEYWORDS && INTEGERS INTO IDENTIFIER

id : OCCURS | INSTANCE | IS_A | HAS_CHILD | HAS_PARENT | LINK
  | SOURCE | SINK | SUBSORT | DOM | ID | MOD | SORT | STATE
  | CONSTRAINTS | FUNCTION | DECLARATIONS | DEFINITIONS | SYSTEM
  | DESCRIPTION | THEORY | MODULE | IMPORT | FROM | DEPENDS
alm_name : id | VAR;

/*
* TERMS denote objects which populate sorts.
* integers are terms .
* true and false are terms.
* variables are terms.
* f(t1, ..., tn) is a term where all ti are terms
* and f is an identifier.
* */

//The pattern for instance and constants of user defined sorts.
object_constant: (id ( '(', term (',' term)* ')')? | integer);
// function_terms and object_constants have the same syntax.
function_term: object_constant;

// terms denote values of sorts.
term: bool | VAR | id | integer | function_term | expression;

var_or_obj: (object_constant | VAR);

/* EXPRESSIONS */
* An expression is a mathematical entity which denotes an integer.
* addition/subtraction
* multiplication/division/modulus/exponent
* factors participate in the above operations.
*/
expression: expression '++' arithmetic_term
            | expression '--' arithmetic_term | arithmetic_term;
arithmetic_term: arithmetic_term '*' factor
                | arithmetic_term '/' factor | arithmetic_term MOD factor
                | factor '^' factor | factor;
factor: VAR | '-' VAR | integer | function_term
       | '-' function_term | '(' expression ')' | '-' '(' expression ')';

// ('-' integer) is not a factor, integer includes NEGINT
/* LITERALS */

//These literals are handled per occurrence.
occurs_atom: OCCURS '(' var_or_obj ')';
instance_atom: INSTANCE '(' var_or_obj ',' sort_name ')';
is_a_atom: IS_A '(' var_or_obj ',' sort_name ')';
link_atom: LINK '(' sort_name ',' sort_name ')';
subsort_atom: SUBSORT '(' sort_name ',' sort_name ')';
has_child_atom: HAS_CHILD '(' sort_name ')';
has_parent_atom: HAS_PARENT '(' sort_name ')';
sink_atom: SINK '(' sort_name ')';
source_atom: SOURCE '(' sort_name ')';

//Pattern for any atom (positive occurrence of a predicate),
// including special atoms.
atom: instance_atom | is_a_atom | link_atom | subsort_atom
    | has_child_atom | has_parent_atom | sink_atom | source_atom
    | function_term ;

//<literal> not including special literals
literal: atom | '-' atom | term relation term ;
occurs_literal: occurs_atom | '-' occurs_atom;

/* ALM FILE */
alm_file : (system_description | theory | module) EOF;

/* ALM SYSTEM DESCRIPTION */

library_name: alm_name;
sys_desc_name: alm_name;
system_description : SYSTEM DESCRIPTION sys_desc_name theory
   (structure solver_mode?)? EOF;    //<system_description>

/* ALM THEORY */

theory_name: alm_name;
theory: (THEORY theory_name sequence_of_modules)
   | (IMPORT theory_name FROM library_name);    //<theory>

/* ALM MODULE */

module_name: alm_name;
   //<set_of_modules><remainder_modules>
sequence_of_modules: (module)+;
   //<module>
module: (MODULE module_name module_body)
   | (IMPORT theory_name (.’’ module_name)? FROM library_name);
module_body: module_dependencies? sort_declarations?
   constant_declarations? function_declarations? axioms?;
/* ALM MODULE DEPENDENCIES */

module_dependencies: DEPENDS ON one_dependency
    (',', one_dependency)*;
one_dependency: (theory_name '.')? module_name;

/* ALM SORT DECLARATIONS */

integer_range: integer '..' integer ;
predefined_sorts: BOOLEANS | INTEGERS | integer_range;
sort_name: predefined_sorts | UNIVERSE | ACTIONS | id ;
new_sort_name : id | integer_range;

//<sort_declaration><remainder_sort_declaration>
sort_declarations: SORT DECLARATIONS (one_sort_decl)+ ;
//<one_sort_decl>,<sort_name>,<remainder_sort_names>,
//<remainder_sorts>
one_sort_decl: new_sort_name (',', new_sort_name)* '::'
    sort_name (',', sort_name)* attributes?;
//<attributes><remainder_attribute_declarations>
attributes: ATTRIBUTES (one_attribute_decl)+;
//<one_attribute_decl>,<arguments>,<remainder_args>
one_attribute_decl: id ':' (sort_name (',', sort_name )*
    RIGHT_ARROW)? sort_name;
/* ALM CONSTANT DECLARATIONS */

//<constant_declarations><remainder_constant_declarations>
constant_declarations: CONSTANT DECLARATIONS (one_constant_decl)+;
//<one_constant_decl>,<const_params>,<remainder_const_params>
one_constant_decl: object_constant (',', object_constant)* ':
    sort_name (',', sort_name)* attribute defs?;

/* ALM FUNCTION DECLARATIONS */

function_name:id;
function_declarations: FUNCTION DECLARATIONS static_declarations?
    fluent_declarations?;
static_declarations: STATICS basic_function_declarations?
    defined_function_declarations?;
fluent_declarations: FLUENTS basic_function_declarations?
    defined_function_declarations?;
basic_function_declarations: BASIC (one_function_decl)+;
defined_function_declarations: DEFINED (one_function_decl)+;
one_function_decl: (TOTAL)? function_name ':.' sort_name
    (('*' sort_name )* RIGHT_ARROW sort_name)?;
pos_fun_def: function_term EQ term | function_term 
   | '-' function_term;

neg_fun_def: function_term NEQ term;

fun_def : (pos_fun_def | neg_fun_def);

/* ALM AXIOMS */

//<axioms>,<remainder_axioms>
axioms: AXIOMS (dynamic_causal_laws | executability_conditions 
   | state_constraints | function_definitions)+ ;
dynamic_causal_laws: DYNAMIC CAUSAL LAWS 
   (one_dynamic_causal_law)*;
executability_conditions: EXECUTABILITY CONDITIONS 
   (one_executability_condition)*;
state_constraints: STATE CONSTRAINTS (one_state_constraint)*;
function_definitions: FUNCTION DEFINITIONS (one_definition)*;

/* DYNAMIC CAUSAL LAW */

//<dynamic_causal_law><body>
one_dynamic_causal_law: occurs_atom CAUSES pos_fun_def IF 
   instance_atom (',', literal)* '.';

/* EXECUTABILITY CONDITION */
//<executability_condition>, <extended body>

one_executability_condition: IMPOSSIBLE occurs_atom IF
  instance_atom (',', (occurs_literal|literal))* '.';

/* STATE CONSTRAINT */

one_state_constraint: fun_def '.' | (FALSE | fun_def) IF
  literal (',', literal)* '.';

/* DEFINITION */

one_definition: function_term '.' | function_term IF
  literal (',', literal)* '.';

/* ALM STRUCTURE */

structure_name: alm_name;
structure: STRUCTURE structure_name (constant_defs
  | instance_defs | statics_defs)*;

/* CONSTANT DEFINITIONS */

//<constant_defs><remainder_constant_defs>
constant_defs: CONSTANT DEFINITIONS (one_constant_def)+;
one_constant_def: object_constant '=' term;
/* INSTANCE DEFINITIONS */

//<instance_defs><remainder_instance_defs>
instance_defs: INSTANCES (one_instance_def)+;
//<one_instance_def>
one_instance_def: var_or_obj (',' var_or_obj)* IN sort_name
     (',' sort_name)* (WHERE literal (',' literal)* )? attribute_defs;
attribute_defs: (one_attribute_def)*;
one_attribute_def: function_term EQ term;

/* STATICS DEFINITIONS */

statics_defs: VALUE OF STATICS (one_static_def)+ ;
one_static_def: fun_def (IF literal (',' literal)* )\' .\';
//<one_static_literal><body>

/* SOLVER MODES */

solver_mode : (temporal_projection | planning_problem
     | diagnostic_problem) added_constraints? action_conditions?;

/* SOLVER MODE COMMON PARTS */

max_steps : MAX STEPS  POSINT;
current_time: CURRENT TIME nat_num;
history : HISTORY (observed | happened)+;
observed : OBSERVED '( function_term , term ,
   nat_num )' '.';

happened : HAPPENED '( object_constant , nat_num )' '.';

/* SOLVER MODE ADDITIONAL CONSTRAINTS */

added_constraints: ADDITIONAL CONSTRAINTS (one_added_constraint)+;
one_added_constraint: (IMPOSSIBLE | AVOID) literal
   (',', literal)* '.';

action_conditions: ACTION (RESTRICTIONS | PERMISSIONS)
   (one_action_condition)+;
one_action_condition: (POSSIBLE | IMPOSSIBLE | AVOID)
   function_term WHEN literal (',', literal)* '.';

/* TEMPORAL PROJECTION SPECIFIC */

temporal_projection : TEMPORAL PROJECTION max_steps history;

/* PLANNING PROBLEM SPECIFIC */

planning_problem : PLANNING PROBLEM max_steps current_time?
   history goal_state;
goal_state: GOAL EQ '{' literal (',', literal)* '}';

/* DIAGNOSTIC PROBLEM SPECIFIC */
diagnostic_problem : DIAGNOSTIC PROBLEM max_steps current_time?

    history normal_conditions? current_state;

normal_conditions: NORMAL CONDITIONS (one_normal_condition)+;

one_normal_condition : id ':' literal ('when' literal

    (',' literal)*)? '.';

current_state: SITUATION EQ '{' literal (',' literal)* '} '.';
Appendix B
CALM User and Developer Manual

B.1 Using CALM

B.1.1 Prerequisites

Required applications for executing CALM.

Java Java JDK version 8 (1.8) or higher.

- Download zip or executable.
- unzip binary into a directory.
- Set JAVA_HOME environment variable to the directory

B.1.2 CALM Distributable

Build the CALM Distributable from source by following the development instructions below or download the Windows distributable from the following url:
https://drive.google.com/open?id=1muof2vvDdJerEaaigVbOMQq6E_s8M7Ku

B.1.3 Commandline Usage

From the CALM\ directory, perform the following command:
java -jar calm.jar <path>\<file>

Where <path>\<file> is the path to a file containing an ALM System description and optional task. The output of the execution will be located in the directory CALM\output\<file>\
B.1.4 Example Applications

If the distributable was downloaded from the above url, an examples directory is included with sample System Descriptions to execute. If the distributable was manually assembled from the development instructions, examples can be copied from the following location:

`ALM-Compiler\src\test\resources\sysdesc\unittest\programs`

Execute the following from within the CALM directory:

`java -jar calm.jar ./examples/basicMotion.tp`

The output of execution will be located at the following directory:

`CALM\output\basicMotion.tp`

B.2 CALM Development

B.2.1 Prerequisites

Required applications for compiling CALM.

Java  Java JDK version 8 (1.8) or higher.


- Download zip or executable.
- If zip, unzip binary into a directory.
- Set JAVA_HOME environment variable to the directory

Maven  Maven version 3.0 or higher.

[https://maven.apache.org/download.cgi](https://maven.apache.org/download.cgi)

- Download zip of binary.
- Unzip binary into a directory.
• Set MAVEN_HOME environment variable to the directory.

• Set M2_HOME environment variable to the directory.

• Add the bin sub-folder to the PATH environment variable.

• Windows:
  https://howtodoinjava.com/maven/how-to-install-maven-on-windows/

• Linux:
  https://www.baeldung.com/install-maven-on-windows-linux-mac

B.2.2 Compiling CALM

From a clean directory on your file system perform the following commands:

> git clone https://github.com/Topology/ALM-Compiler.git
> cd ./ALM-Compiler
> mvn clean build package

The compiled jar with all dependencies is in the target sub-directory.

B.2.3 Creating The CALM Distributable

After compiling the maven project (mvn clean build package) follow the following steps to create a distributable directory for the intended target operating system.

1. Create a new empty directory called CALM\.

2. Copy and rename the alm compiler jar containing its dependencies from
   ALM-Compiler\target\alm-compiler-...-with-dependencies.jar
   to the new directory: CALM\calm.jar

3. Create a subfolder CALM\clingo\.
4. Download and unzip the appropriate OS specific version of clingo from 
https://sourceforge.net/projects/potassco/files/clingo/4.5.4/ 
into the CALM\clingo\ directory.

5. Rename the clingo executable to clingo or clingo.exe as required by your
operating system. The program CALM\clingo\clingo should be executable.

6. Create a new folder CALM\sparc\ 

7. Download the sparcl.jar from
https://github.com/iensen/sparcl/raw/master/sparcl.jar
into the directory and verify that java -jar CALM\sparc\sparcl.jar executes.

8. Create a subfolder CALM\output\ 

9. Create a subfolder CALM\library\ 

10. Add any libraries and theories under the following pattern:
    CALM\library\<library_name>\<theory_name>.alm
    where the file <theory_name>.alm contains an ALM Theory with the same
        name as the file name.

B.2.4 CALM Architecture

The organization of the source code is presented here through describing the contents
of important directories and files.

ALM Grammar and Parser ALM-Compiler\src\main\antlr4\ALM.g4

- This file contains the ALM grammar encoded in ANTLR4’s input language.
  After modifying the grammar, this file is given as input to the ANTLR4 tool
to generate a new parser for the modified ALM grammar.
After modifying the ALM grammar, the parser is created in 3 steps:

1. Compile the project to generate the ANTLR4 parser from the grammar.

2. Copy the generated code (except ALMBaseListener.java) to the right location.

3. Update the previous ALMBaseListener.java implementation.

We recommend developing CALM using an IDE. Changes to the grammar will alter the generated parser in complex ways. Using an IDE will track changes between the new ALMLListener.java interface and the old ALMBaseListener.java implementation of the interface.

**ALM-Compiler\target\generated-sources\antlr4\**

- Contains the generated source code for the ANTLR4 parser after the maven project has been compiled: `mvn clean build`.
  
  (Do not copy the ALMBaseListener.java.)

**ALM-Compiler\src\main\java\edu\ttu\krlab\alm\parser**

- Copy the generated ANTLR4 parser (except ALMBaseListener.java) to this location. Open the project in an IDE to edit the old copy of ALMBaseListener.java. The IDE will indicate the places where the old ALMBaseListener is incompatible with the new parser and ALMLListener interface.

The Main Class and Compiler Configuration

**ALM-Compiler\src\main\java\edu\ttu\krlab\alm\ALMCompiler.java**

- This file contains the main function which processes commandline arguments, configures the CALM settings, parses the input ALM System Description, and translates the system description to SPARC.
ALM-Compiler\src\main\java\edu\ttu\krlab\alm\ALMCompilerSettings.java

- This file contains the configuration of CALM and processing of commandline arguments.

Multi-Module BAT Hierarchy

ALM-Compiler\src\main\java\edu\ttu\krlab\alm\ALMModuleManager.java

- This file keeps track of the references to libraries and modules and resolves a module reference to the portion of the symbol table created by parsing the module definition. The \texttt{resolveModules()} function recursively imports modules from libraries on disk until all module references are resolved or there is a module resolution failure.

ALM-Compiler\src\main\java\edu\ttu\krlab\alm\datastruct\sig\`

- This directory contains the SymbolTable.java class, which models a modular or hierarchical BAT signature. It also contains the class definitions needed to model elements within the BAT Signature.

Semantics and Error Checking

ALM-...\src\main\java\edu\ttu\krlab\alm\parser\ALMBaseListener.java

- This file contains the implementations of the \texttt{enter} and \texttt{exit} functions called before and after parsing each non-terminal in the ALM grammar. The \texttt{exit} function is provided the syntax tree produced by the ANTLR4 parser. Through implementing the non-terminal \texttt{exit} functions of key grammatical elements of ALM we perform error checking, construction of the BAT Symbol Table, and creation of $\mathcal{ASP}\{f\}$ rules for user defined and auxiliary axioms.
This class assists with parsing and translating ANTLR4 syntax tree objects to terms in ALM. It also assists with type checking nested terms.

This directory contains the ErrorReport.java class for reporting errors during compilation and the ErrorMessageTable.java class which defines the messages to produce when reporting each error.

This directory contains the TypeChecker.java class which is used to track variable occurrences in $\text{ASP}\{f\}$ rules and check that a consistent type or sort can be inferred between all variable occurrences.

This directory contains SPARCProgram.java class and other elements needed to model the elements of a $\text{SPARC}$ Program.

This class contains the functions which translate the BAT Signature in the symbol table to the sorts section, translate the function signatures in the symbol table to the predicate section, and translate the $\text{ASP}\{f\}$ rules to $\text{ASP}$ rules in the rules section of the $\text{SPARC}$ program.
Interacting with the $\text{SPARC}$ Solver

ALM-Compiler\src\main\java\edu\ttu\krlab\answerset\parser\

- This directory contains classes and utilities needed to execute the SPARC solver and process the resulting answer sets.
Appendix C

IQTD Proofs

C.1 Extra Definitions

Given an IQ problem $\langle \Pi, C \rangle$, let $I = \langle I_0, I_1, \ldots, I_n \rangle$ be the IQ sequence defined by $C$. $\langle \Pi, C \rangle$ is called a finite IQ problem if for $i \in [1..n]$ the sld-derivation tree for $\langle \text{flatten}(I_i) | \emptyset \rangle$ is finite with respect to $\Pi$.

Given an IQ state $S = \langle \langle I_A, I_D \rangle, N, R, P, (Q|C), M \rangle$, let $n = |I_A \sim I_D|$, $m = |I_A|$, and $R = \langle T_1, \ldots, T_n \rangle$. The rank of $S$, denoted $\text{rank}(S)$, is defined as follows:

- If $M = \text{halt}$, let $P'$ be the last path of computation in $T_n$. $\text{rank}(S) = \text{rules}(P')$
- If $M = \text{proceed}$, let $P'$ be the first open path of computation in $T_{m-1}$. $\text{rank}(S) = \text{rules}(\text{exp}(P', I_A[[I_A]]) \sim P)$
- If $M = \text{backtrack}$, let $P'$ be the first open path of computation in $T_{m-1}$. $\text{rank}(S)$ is the greatest path of derivation in the sld-derivation tree of $\langle \text{flatten}(I_A \sim I_D) | \emptyset \rangle$ which has $\text{rules}(\text{exp}(P', I_A[[I_A]]) \sim P)$ as a prefix.

Given a CLP Program $\Pi$, a path $P = \langle s_0, e_1, s_1, \ldots, e_z, s_z \rangle$ in IQTD($\Pi$) is called a run when $s_0$ and $s_z$ are halted IQ states and for $i \in [1..z-1]$, $s_i$ is not a halted IQ state.

Given a CLP Program $\Pi$ and a run $P = \langle s_0, e_1, s_1, \ldots, e_z, s_z \rangle$ in IQTD($\Pi$), the IQ Command of the run is the IQ command labeling the first IQ State transition of $P$. Note that from the definition of the IQ State transitions, the first transition must be an IQ Command Transition and there can be no other IQ Command Transitions in the run.
Given a CLP Program Π and a halted sound path \( P = \langle s_0, e_1, s_1, \ldots, e_z, s_z \rangle \) in \( IQTD(\Pi) \), let \( L = \langle L_0, \ldots, L_n \rangle \) be the halted IQ states along \( P \) where \( s_0 = L_0 \) and \( S_z = L_N \), for \( i \in \{1..n\} \) the \( i_{th} \) run of \( P \) is the path from \( L_{i-1} \) to \( L_i \), the \( i_{th} \) IQ Command of \( P \) is the IQ Command \( C_i \) labeling the first IQ State Transition in the \( i_{th} \) run, and the sequence \( \langle C_1, \ldots, C_n \rangle \) is called the Sequence of IQ Commands defined by \( P \).

**C.2 Deterministic Run Lemma**

Given an IQ command \( C' \) and a halted sound path \( P \) in \( IQTD(\Pi) \) where \( C = \langle C_1, \ldots, C_n \rangle \) is the sequence of IQ commands defined by \( P \) and \( L = \langle L_0, L_1, \ldots, L_n \rangle \) is the sequence of halted IQ states in \( P \), if \( \langle \Pi, C \sim (C') \rangle \) is a finite IQ problem then there exists a unique run \( \langle s_0, e_1, s_1, \ldots, e_z, s_z \rangle \) in \( IQTD(\Pi) \) such that \( s_0 = L_n \) and \( e_1 = C' \).

**Proof**

Assume \( C' \) is an IQ command and \( P \) is a halted sound path in \( IQTD(\Pi) \). (3)

Let \( C = \langle C_1, \ldots, C_n \rangle \) be the IQ commands along \( P \), \( L = \langle L_0, L_1, \ldots, L_n \rangle \) be the sequence of halted IQ states along \( P \), and \( I = \langle I_0, I_1, \ldots, I_n, I_{n+1} \rangle \) be the IQ sequence defined by \( \langle C_1, \ldots, C_n, C' \rangle \).

Assume \( \langle \Pi, C \sim (C') \rangle \) is a finite IQ problem. (4)

We construct a path \( P' \) in \( IQTD(\Pi) \) as follows:

1. If \( \langle L_n, C', s_1 \rangle \) is an IQ command transition in \( IQTD(\Pi) \) then
   \[
P' := \langle L_n, C', s_1 \rangle
   \]

2. Loop
   (a) \( P' \) is of the form \( \langle L_n, C', s_1, \ldots, e_j, s_j \rangle \)
(b) If there exists an IQ state transition \( \langle s_j, e_{j+1}, s_{j+1} \rangle \) in IQTD(\( \Pi \)) that is not an IQ command transition then

\[
P' := \langle L_n, C', s_1, \ldots, e_j, s_j, e_{j+1}, s_{j+1} \rangle
\]

(c) otherwise exit the loop.

From the definition of all state transitions, each non-halted state \( s' \) along \( P' \) after \( L_n \) has a unique transition \( \langle s', e, s'' \rangle \).

Since the IQ problem is finite (4), there are a finite number of state transitions in \( P' \) and \( P' \) ends in a halted IQ state.

\( P' \) is the only run starting with the IQ command transition \( \langle L_n, C', s_1 \rangle \).

\( \square \)

C.3 The Correspondence Lemma

The first element, \( \langle I_A, I_D \rangle \), of a IQ state is called the query of the state.

Given a halted sound path \( P = \langle s_0, e_1, \cdots, e_z, s_z \rangle \), for any IQ state \( s_i \), we define \( c(i) \) as the maximal number such that

- \( c(i) \leq i \), and
- \( C_{c(i)} \) is an IQ command.

We call \( C_{c(i)} \) the IQ command for \( s_i \).

For any state \( s_i \) and \( s_j \) \( (j \geq i) \) of \( P \), let \( \langle I_A, I_D \rangle \) be the query of \( s_i \). we say \( s_i \) is useful for \( s_j \) if for all \( m \in [c(i), j] \), \( I_A \) is a prefix of \( I_m \) let \( I_m \) be the IQ sequence defined by \( \langle C_1, \cdots, C_{c(m)} \rangle \).

Given a halted sound path \( P \) in IQTD(\( \Pi \)) let

- \( P \) be of the form \( \langle s_0, e_1, s_i, \ldots, e_z, s_z \rangle \)
• $C = \langle C_1, \ldots, C_n \rangle$ be the sequence of IQ commands defined by $P$.

• $I = \langle I_0, \ldots, I_n \rangle$ be the IQ Sequence defined by $C$.

• for $i \in [1..n]$ and $g \in [1..|I_i|]$, let $m_{i,g}$ be the number of solutions for incremental query $I_i[1..g]$

• $L = \langle L_0, \ldots, L_n \rangle$ be the sequence of halted states in $P$

• for $k \in [1..n]$, let $P_k = \langle s_{l_k} = L_{k-1}, e_{l_k+1} = C_k, s_{l_k+1}, \ldots, e_{l_k+i}, s_{l_k+i} = L_k \rangle$ be the $k_{th}$ run of $P$

• for $i \in [0..z]$, let $\langle I_{A_i}, I_{D_i}, R_i, N_i, P_i, \langle Q_i | C_i \rangle, M_i \rangle$ be the form of each IQ state $s_i$.

For $j \in [0..z]$, let $k$ be the least integer such that run $P_k$ contains $s_j$, then the following holds:

• If $s_j$ is the first IQ state in $P_k$ then $I_{A_j} \sim I_{D_j} = I_{k-1}$, otherwise $I_{A_j} \sim I_{D_j} = I_k$. 
(CL.1)

• For $i \in [1..|I_k|]$, for $g \in [1..|R_j[i]|]$, $R_j[i][g]$ is a successful path of computation for $\langle flatten(I_k[1..i]) | \emptyset \rangle$ and the annotated solution derived from $R_j[i][g]$ is the $g_{th}$ annotated solution of incremental query $I_k[1..i]$ 
(CL.2)

• If $C_k = dec()$ then let $q = |I_k|$, otherwise let $q = |I_k - 1|$. For $g \in [1..q]$ if $N_j[g] < m_{k,g}$ then $N_j[g] \leq |R_j[g]| \leq m_{k,g} \text{ otherwise } |R_j[g]| = m_{k,g}$.
(CL.3)

• If $C_k = inc(Q)$ and $s_j$ is not the first IQ State of $P_k$, let $q = |I_k|$, then $N_j[q] = 1$. If $s_j$ is the last IQ State in $P_k$ and $m_{k,q} > 0$ then $|R_j[q]| = 1 \text{ otherwise } |R_j[q]| = 0$. (CL.4)
• If $C_k = \text{next}()$, $I_k \neq I_0$, and $s_j$ is not the first IQ State of $P_k$, let $h$ be the position of the related increment query command for $C_k$, let $r$ be the number of solution requests for $I_k$ in $\langle C_h, \ldots, C_k \rangle$, let $q = |I_k|$, then $N_j[q] = r$. If $r > m_{k,q}$ then $|R_j[q]| = m_{k,q}$. If $r \leq m_{k,q}$ and $s_j$ is the last IQ State of $P_k$ then $r \leq |R_j[q]| \leq m_{k,q}$. If $r \leq m_{k,q}$ and $s_j$ is not the last IQ State of $P_k$ then $r - 1 \leq |R_j[q]| \leq m_{k,q}$.  

(CL.5)

• If $M_j \neq \text{halt}$, let $q = |I_A_j|$ and $P$ be defined as follows: If $q = 1$ then $P = P_j$. If $q > 1$ then $P = \exp(P_o, I_A_j[q]) \leadsto \langle P_j \rangle$ where $P_o$ is the first open path of $R_j[q - 1]$. Let $\langle F_1, \ldots, F_m \rangle$ be the sequence of choice frames along $P$. Let $T = \langle (V, E), L_V, L_E \rangle$ be the computation tree for $\langle \text{flatten}(I_A_j) \rangle \emptyset$. For $a \in [1..m]$ there exists a node $b_a \in V$ such that $(F_1, \ldots, F_a)$ is the path of computation to $b_a$ in $T$ and $L_V(b_a) = \text{con}^*(\text{res}(\langle Q_a | C_a \rangle, r_a))$ where $F_a = \langle \langle Q_a | C_a \rangle, \{r_a \}, L_a \rangle$. If $a = m$ and $M_j = \text{proceed}$ then $L_V(b_a) = \text{con}^*(\langle Q_j | C_j \rangle)$.  

(CL.6)

• If $j > 0$ and $\langle s_{j-1}, e_j, s_j \rangle$ is not an increment query or decrement query command transition, then $\text{rank}(s_j) \geq_{\omega} \text{rank}(s_{j-1})$. For $i \in [1..|I_k|]$ Let $T_{\text{sl}d_i} = \langle (V_{\text{sl}d_i}, E_{\text{sl}d_i}), L_{V_{\text{sl}d_i}}, L_{E_{\text{sl}d_i}} \rangle$ be the sld-derivation tree for $\langle \text{flatten}(I_k[1..i]) \rangle \emptyset$ and for all $b \in V_{\text{sl}d_i}$ let $D_b$ be the path of derivation to $b$ in $T_{\text{sl}d_i}$. If $D_b \leq_{\omega} \text{rank}(s_j)$ there exists $s_x$ such that $s_x$ is useful for $s_j$, $\text{rank}(s_x) = D_b$, and if $D_b$ is a successful path of derivation then $\langle s_x, e, s_{x+1} \rangle$ is either a intermediate save transition or a final save transition.  

(CL.7)

• If $P_j \neq \square$ then for $i \in [1..|P_j|]$ let $\langle \langle Q_i | C_i \rangle, \{r_i \}, L_i \rangle = P_j[i]$ and let $L_i^*$ be the set of rules from $\Pi$ which resolve with $\langle Q_i | C_i \rangle$. The following properties hold: $r_i \in L_i^*$ and $\forall r^* \in L_i^*, r^* \in L_i \iff r^* >_{\omega} r_i$.  

(CL.8)

Proof
Let the premises of the correspondence lemma be given. We prove that all claims (CL.1) through (CL.8) hold through strong induction on the length of a prefix of 

\[ P = \langle s_0, e_1, \cdots, e_z, s_z \rangle. \]

It is clear that all claims hold for \( s_0 \), the initial IQ halted state.

Assume for \( n \in [1..z - 1] \) that all claims hold for all states \( s_i \) such that \( i \in [1..n] \), we show by cases that each claim holds for \( s_{n+1} \). (5)

Proof of claim (CL.1):

Suppose \( s_{n+1} \) is in the \( k_{th} \) run of \( P \).

case 1: \( s_{n+1} \) is the first IQ state of \( P_k \)

\( s_{n+1} \) is the last IQ state of \( P_{k-1} \)

\[ I_{A_{n+1}} \sim I_{D_{n+1}} = I_{k-1} \] holds by (5) for claim (CL.1)

case 2: \( s_{n+1} \) is not the first IQ state of \( P_k \)

Let \( \langle s_t, C_k, s_{t+1} \rangle \) be the first state transition of \( P_k \).

by case 1 \( I_{A_t} \sim I_{D_t} = I_{k-1} \)

by the definition of all IQ Command Transitions, \( I_{A_{t+1}} \sim I_{D_{t+1}} = I_k \)

by the definition of all state transitions that are not IQ Command Transitions, for all states \( s_i \in [s_{t+1}, \ldots, s_{n+1}] \) in \( P_k \), \( I_{A_i} \sim I_{D_i} = I_k \)

therefore \( I_{A_{n+1}} \sim I_{D_{n+1}} = I_k \)

claim (CL.1) is proven.

Proof of claim (CL.2):

\[ \langle s_n, e_{n+1}, s_{n+1} \rangle \] is not a save or a command transition.
(CL.2) holds by strong inductive hypothesis for all states prior to $s_{n+1}$ for IQ state transitions that are not save or command transitions, the record of computation and the incremental queries of $s_{n+1}$ and $s_n$ are the same.

Claim (CL.2) is satisfied in this case.

case 2: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is an increment command transition

$I_{k-1}$ is a proper sub-incremental query of $I_k$ $|I_{k-1}| < |I_k|$

$R_{n+1} = R_n \prec \langle \rangle$

Claim (CL.2) is satisfied in this case.

case 3: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is a next command transition

The incremental query and record of computation remain unchanged between $s_n$ and $s_{n+1}$

by the assumption of strong inductive hypothesis (CL.2) holds for $s_n$

Claim (CL.2) for $s_{n+1}$ is satisfied in this case.

case 4: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is a decrement command transition

$I_k$ is a proper sub-incremental query of $I_{k-1}$

$R_{n+1}$ is a proper prefix of $R_n$

by the assumption of strong inductive hypothesis (CL.2) holds for $s_n$

Claim (CL.2) holds for $I_k$ and $R_{n+1}$ in $s_{n+1}$

case 5: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is a save transition

The incremental queries of $s_n$ and $s_{n+1}$ are the same.

$R_{n+1}$ is the same as $R_n$ except that $R_{n+1}[|I_{A_n}|] = R_n[|I_{A_n}|] \prec \langle \langle P' \rangle \rangle$ where $P'$ is a successful path of computation to a solution of $flatten(I_{A_n})$.  

135
Let $g$ be such that $R_{n+1}[|I_{A_n}||g] = P'$.

The rank or $P'$ is greater than all other saved paths in $R_{n+1}$ by the assumption of strong inductive hypothesis for claim (CL.7)

By the assumption of strong inductive hypothesis for claim (CL.8), the rank of $P'$ is less than all remaining unexplored paths.

By the definition of all state transitions, resolution always picks the least rule to resolve with the remaining query and no branch in the sld-derivation tree is skipped.

$R_{n+1}[|I_{A_n}||g] = P'$ is the $gh$ annotated solution for $flatten(I_{A_n})$.

claim (CL.2) for $s_{n+1}$ is satisfied in this case.

claim (CL.2) is proven.

Proof of claim (CL.3):

by strong inductive hypothesis (CL.3) is satisfied for $s_n$.

by the definition of each state transition, (CL.3) is satisfied in each case.

claim (CL.3) is proven.

Proof of claim (CL.4):

let $P_k$ be the run containing the transition $\langle s_n, e_{n+1}, s_{n+1} \rangle$.

By the deterministic run lemma and the fact that the IQ problem is finite, $P_k$ is finite in length.

Assume $C_k$ is an increment command.

For all states along $P_k$ after the command transition $|R_{n+1}[|I_{A_{n+1}}]| = 0$
\( P_k \) either ends in a final save transition and \(|R_{n+1}[I_{A_{n+1}}]| = 1\) or ends in a fail transition and \(|R_{n+1}[I_{A_{n+1}}]| = 0\).

claim (CL.4) is proven.

Proof of claim (CL.5):

let \( P_k \) be the run containing the transition \( <s_n, e_{n+1}, s_{n+1}> \)

let \( q = |I_k| \).

By the deterministic run lemma and the fact that the IQ problem is finite, \( P_k \) is finite in length.

Assume \( C_k \) is a next command and \( s_x \) is the first IQ state in \( P_k \) such that \( x < n+1 \).

For all states \( s_i \) along \( P_k \) after the command transition and before the final state \(|R_i[q]| = |R_x[q]|\) and \( N_i[q] = N_x[q] + 1\).

by the assumption of strong induction, claim (CL.5) holds for all states along \( P_k \) prior to \( s_{n+1} \).

case 1: \( <s_n, e_{n+1}, s_{n+1}> \) is not the final transition in \( P_k \), claim (CL.5) holds.

case 2: \(|R_n[q]| = m_{k,q}\) and \( <s_n, e_{n+1}, s_{n+1}> \) is the final transition in \( P_k \).

All solutions have been previously found for flatten(\( I_k \)).

by the assumption of strong induction, all claims hold for all states prior to \( s_{n+1} \).

from claim (CL.7) the paths of derivation in the sld-derivation tree are explored in a monotonically increasing manner.

since the last annotated solution has been previously recorded, all paths of derivation of higher rank do not lead to new solutions.
$P_k$ ends in a fail transition.

$|R_{n+1}[q]| = m_{k,q}$ and $N_{n+1}[q] = N_n[q]$.

claim (CL.5) holds for $s_{n+1}$.

case 3: $|R_n[q]| < m_{k,q}$ and $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is the final transition in $P_k$.

$\langle s_n, e_{n+1}, s_{n+1} \rangle$ is either a fail transition or a final save transition.

by the assumption of strong induction, all claims hold for all states prior to $s_{n+1}$.

from claim (CL.7) the paths of derivation in the sld-derivation tree are explored in a monotonically increasing manner.

from claim (CL.8) the paths of derivation are explored in order.

since not all solutions have been recorded, there are successful paths of derivation remaining to be explored.

$\langle s_n, e_{n+1}, s_{n+1} \rangle$ must be a final save transition.

From the definition of all transitions along $P_k$, $N_{n+1}[q] = N_n[q] = N_x[q] + 1$ and $|R_{n+1}[q]| = |R_x[q]| + 1$

since claim (CL.5) holds for $s_x$, claim (CL.5) holds for $s_{n+1}$.

claim (CL.5) is proven.

Proof of claim (CL.6):

We prove claim (CL.6) by examining modes of the solver as it explores the sld-derivation tree and constructs paths of computation. This claim asserts that for every non-halted state of the solver, there is a well defined node in the computation tree for the active incremental query that the solver is visiting.

case 1: $M_{n+1} = proceed$
by assumption of strong inductive hypothesis, claim \((\text{CL.6})\) holds for \(s_n\).

case 1.1: \(\langle s_n, e_{n+1}, s_{n+1} \rangle\) is a resolution transition.

Let \(b\) be the node in the computation tree of \(\langle \text{flatten}(I_{A_n}), \emptyset \rangle\) for state \(s_n\).

Since \(\langle s_n, e_{n+1}, s_{n+1} \rangle\) is a resolution transition, there exists rules which resolve with the head of the active CLP query.

let \(r\) be the rule chosen for resolution by this transition.

let \(b_r\) be the node in the computation tree annotated by the choice frame reflecting resolution with \(r\).

The existence of \(b_r\) satisfies claim \((\text{CL.6})\) in this case.

case 1.2: \(\langle s_n, e_{n+1}, s_{n+1} \rangle\) is a constraint transition.

Let \(b\) be the node in the computation tree of \(\langle \text{flatten}(I_{A_n}), \emptyset \rangle\) for state \(s_n\).

Since \(\langle s_n, e_{n+1}, s_{n+1} \rangle\) is a constraint transition, \(b\) is the node in the computation tree satisfying claim \((\text{CL.6})\) in this case.

case 1.3: \(\langle s_n, e_{n+1}, s_{n+1} \rangle\) is an intermediate save transition.

Note that the active path of computation in \(s_{n+1}\) is an empty extension of the active path of computation in \(s_n\).

Let \(b\) be the node in the computation tree of \(\langle \text{flatten}(I_{A_n}), \emptyset \rangle\) for state \(s_n\) by strong inductive hypothesis.

\(b\) is the node in the computation tree satisfying claim \((\text{CL.6})\) in this case.

case 2: \(M_{n+1} = \text{backtrack}\)

Note that proceed transitions build the active path of computation one choice frame at a time.

The sequence of choice frames along active path in a state whose mode is backtrack has been previously active in a proceed transition. Let \(s_x\) be the proceed transition prior to \(s_{n+1}\) which had the same active path.
by assumption in strong induction, claim (CL.6) holds for \( x \).

claim (CL.6) holds for \( s_{n+1} \)

claim (CL.6) is proven.

Proof of claim (CL.7):

The essence of this claim is that the sld-derivation trees are searched and paths of computation are constructed in a monotonically increasing order with respect to rank and that all solutions belonging to paths of derivation of lower rank have been saved in the record of computation.

We prove claim (CL.7) in cases of the \( \langle s_n, e_{n+1}, s_{n+1} \rangle \) transition.

by assumption of strong inductive hypothesis, claim (CL.6) holds for all states \( s_i \) where \( i < n \).

case 1: \( \langle s_n, e_{n+1}, s_{n+1} \rangle \) is a next command transition, constraint transition or save transition.

By the definition of rank and the definition of the transition, \( rank(s_{n+1}) = \omega \)

\( rank(s_n) \)

Since \( s_{n+1} \) is of the same rank as \( s_n \) and claim (CL.7) holds for \( s_n \), claim (CL.7) holds for \( s_{n+1} \).

case 2: \( \langle s_n, e_{n+1}, s_{n+1} \rangle \) is a resolution transition.

Let \( P'_n \) be the active path of computation for state \( s_n \)

Let \( P'_{n+1} \) be the active path of computation for state \( s_{n+1} \)

By definition of a resolution transition \( P'_{n+1} \) extends \( P'_n \) and \( rules(P'_n) < \omega \)

\( rules(P'_{n+1}) \)
Since $P'_n$ is a prefix of $P'_{n+1}$ and resolution selected the least rule to resolve with, there does not exist a path of derivation $P_b$ in the sld-derivation tree such that $\text{rules}(P'_n) <_\omega P_b <_\omega \text{rules}(P'_{n+1})$.

Since claim (CL.7) holds for $s_n$ by strong inductive hypothesis, (CL.7) holds for $s_{n+1}$ in this case.

case 3: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is any backtrack transition except for the fail transition.

Let $P'$ be the active path of computation for state $s_{n+1}$

Let $x < n+1$ be the largest integer such that $\langle s_{x-1}, e_x, s_x \rangle$ is a proceed transition and $P'$ is the active path of computation for state $s_x$

By definition of rank, $\text{rank}(s_x) = \text{rank}(s_{n+1})$

by assumption of strong inductive hypothesis, claim (CL.7) holds for $s_x$

claim (CL.7) holds for $s_{n+1}$ in this case.

case 4: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is any backtrack transition except for the fail transition.

Let $P'$ be the active path of computation for state $s_{n+1}$

Let $x < n+1$ be the largest integer such that $\langle s_{x-1}, e_x, s_x \rangle$ is a proceed transition and $P'$ is the active path of computation for state $s_x$

By definition of rank, $\text{rank}(s_x) = \text{rank}(s_{n+1})$

by assumption of strong inductive hypothesis, claim (CL.7) holds for $s_x$

claim (CL.7) holds for $s_{n+1}$ in this case.

case 5: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is a decrement command transition or fail transition.

Let $P'$ be the last saved successful path of computation in $R_{n+1}[|I_{A_{n+1}}|]$.

Let $x < n + 1$ be the largest integer such that $\langle s_{x-1}, e_x, s_x \rangle$ is the final save transition which add $P'$ to $R_{n+1}$.
By definition of rank, $rank(s_x) = rank(s_{n+1})$

by assumption of strong inductive hypothesis, claim (CL.7) holds for $s_x$

claim (CL.7) holds for $s_{n+1}$ in this case.

case 6: $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is an increment command transition.

Let $P'$ be the first saved successful path of computation in $R_{n+1}[|I_{A_{n+1}}| - 1]$.

Let $x < n + 1$ be the largest integer such that $\langle s_{x-1}, e_x, s_x \rangle$ is the save transition which add $P'$ to $R_x$.

By definition of rank, $rank(s_x) = rank(s_{n+1})$

by assumption of strong inductive hypothesis, claim (CL.7) holds for $s_x$

claim (CL.7) holds for $s_{n+1}$ in this case.

claim (CL.7) is proven.

Proof of claim (CL.8):

The essence of claim (CL.8) is that in each application of the resolution transition and the backtrack within current query transitions, the construction of the choice frame is such that the least available rule is chosen for resolution. This ensures that the sld-derivation tree is explored in order with respect to the $<_{\omega}$ ordering on paths of derivation.

This property is trivially satisfied by strong inductive hypothesis for all transitions $\langle s_n, e_{n+1}, s_{n+1} \rangle$ other than the resolution transition and the backtrack within the current query transition.

When $\langle s_n, e_{n+1}, s_{n+1} \rangle$ is either the resolution transition or the backtrack within the current query transition, this claim is satisfied by construction of the active path of computation in the definition of the transitions.
claim (CL.8) is proven.

By strong induction, The Correspondence Lemma holds for all states along every sound path in the IQTD transition diagram.

□

C.4 Proof Of Theorem

**Theorem** Given a finite IQ problem \( \langle \Pi, C \rangle \), \( S \) is an IQ solution to \( \langle \Pi, C \rangle \) if and only if there exists a halted sound path \( P \) in \( IQTD(\Pi) \) such that \( C \) is the sequence of IQ commands defined by \( P \) and \( S \) is the IQ solution defined by \( P \).

**Proof** \( \Leftarrow \) (uses Correspondence Lemma)

**Claim:** Given an IQ problem \( \langle \Pi, C = \langle C_1, \ldots, C_n \rangle \rangle \), \( S \) is an IQ solution to \( \langle \Pi, C \rangle \) if there exists a halted sound path \( P \) in \( IQTD(\Pi) \) such that \( C \) is the sequence of IQ commands defined by \( P \) and \( S \) is the IQ solution defined by \( P \).

**Proof:**

Assume \( P \) is a halted sound in \( IQTD(\Pi) \) where \( C \) is the sequence of IQ commands in \( P \)

Let \( S \) be the IQ solution defined by \( P \)

Let \( L = \langle L_0, \ldots, L_n \rangle \) be the sequence of halted IQ states along \( P \), \( C \) be \( \langle C_1, \ldots, C_n \rangle \), \( S \) be \( \langle S_1, \ldots, S_n \rangle \), \( I = \langle I_0, \ldots, I_n \rangle \) be the IQ Sequence defined by \( C \), and for \( i \in [1..n] \), let \( L_i \) be \( \langle \langle I_{A_i}, I_{D_i} = \langle \rangle \rangle, R_i, N_i, \square, \langle \square | \square \rangle, \text{halt} \rangle \).

To prove that \( S \) is the IQ Solution for the IQ Problem \( \langle \Pi, C \rangle \), for \( i \in [1..n] \), we consider three cases of the IQ Command \( C_i \) in the \( i_{th} \) run \( \langle L_{i-1}, C_i, \ldots, L_i \rangle \) of \( P \) which defines \( S_i \).

Case 1: \( C_i = inc(Q) \). We consider two possibilities for \( I_i \) in our proof of 16
Case 1.1: No solution exists for $I_i$ has

From $C_i = \text{inc}(Q)$ we have that $C_i \neq \text{dec}()$. (6)

By the definition of IQ Sequence defined by $C$ and $C = \text{inq}(Q)$, $I_i$ contains the query $Q$ and $|I_i| > 0$.

By the correspondence lemma (CL.1), since $L_i$ is in the $i_{th}$ run of $P$ and not the first halted state in the run, $I_i = I_{A_i} \bowtie I_{D_i}$.

Since $I_{D_i} = \langle \rangle$ we have that $I_i = I_{A_i}$ and $|I_{A_i}| > 0$. (7)

By the correspondence lemma (CL.4), since $L_i$ is in the $i_{th}$ run of $P$ and is not the first halted state of the run and $C_i = \text{inc}(Q)$ we have $N_i[k] = 1$.

By the correspondence lemma (CL.4), since $I_i$ is the incremental query of $L_i$ and $I_i$ has 0 solutions, and $0 < N_i[k] = 1$, we conclude that $|R_i[k]| = 0$.

Therefore $|R_i[k]| < N_i[k]$ (8)

From the definition of IQ Solution defined by $P$, if $C_i \neq \text{dec}()$, $|I_{A_i}| = k > 0$ and $|R_i[k]| < N_i[k]$, then $S_i = \text{fail}$.

Since 6, 7 and 8 hold, we conclude that:

$S_i = \text{fail}$ (9)

Case 1.2: $I_i$ has at least 1 solution

By the definition of IQ Sequence defined by $C$ and $C_i = \text{inc}(Q)$, $I_i$ contains the query $Q$ and $|I_i| > 0$.

Let $|I_i| = k$ for some $k > 0$.

By the correspondence lemma (CL.1), since $L_i$ is in the $i_{th}$ run of $P$ and not the first halted state in the run, $I_i = I_{A_i} \bowtie I_{D_i}$.

Since $I_{D_i} = \langle \rangle$ we have that $I_i = I_{A_i}$ and $|I_{A_i}| = k > 0$. (10)
From $C_i = \text{inc}(Q)$ we have $C_i \neq \text{dec}$. \hspace{1cm} (11)

By the correspondance lemma (CL.4), since $L_i$ is in the $i_{\text{th}}$ run of $P$ and is not the first halted state of the run and $C_i = \text{inc}(Q)$ we have $N_i[k] = 1$.

By 10 $I_i$ is the incremental query of $L_i$. \hspace{1cm} (12)

$I_i$ has at least 1 solution and $L_i$ is a halted IQ state and $1 \geq N_i[k] = 1$. \hspace{1cm} (13)

By 12, 13 and correspondance lemma (CL4) we have $|R_i[k]| = N_i[k]$ and then we have $|R_i[k]| \geq N_i[k]$. \hspace{1cm} (14)

By 11, 14 and from the definition of IQ Solution defined by $P$, $S_i$ is the annotated solution derived from $R_i[k][N_i[k]] = R_i[k][1]$.

By the correspondance (CL.2), the annotated solution derived from $R_i[k][1]$ is the $k_{\text{th}}$ annotated solution of incremental query $I_i[1..k]$.

Since $k = |I_i|$ therefore $I_i[1..k]$ is $I_i$.

$S_i$ is the first annotated solution for $I_i$. \hspace{1cm} (15)

Combining 9 and 15 we have that when $C_i = \text{inc}(Q)$, if no solution exists for $I_i$, then $S_i = \text{fail}$, otherwise $S_i$ is the first annotated solution for $I_i$. \hspace{1cm} (16)

Case 2: $C_i = \text{next}()$. We consider two possibilities for $|I_i|$ in our proof of 35.

Case 2.1: $|I_i| = 0$

By the correspondance lemma (CL.1), since $L_i$ is in the $i_{\text{th}}$ run of $P$ and not the first halted state in the run, $I_i = I_{A_i} \cap I_{D_i}$.

Since $I_{D_i} = \langle \rangle$ we have that $I_i = I_{A_i}$. \hspace{1cm} (17)

By 17 and $|I_i| = 0$ we have $|I_{A_i}| = 0$. \hspace{1cm} (18)

By 18, $C_i = \text{next}()$ and from the definition of IQ Solution defined by $P$, if $C_i = \text{next}()$ and $|I_{A_i}| = 0$ then $S_i = \Box$.  

145
Case 2.2: $|I_i| > 0$. We consider two possibilities for $I_i$ in our proof of 34.

Before start to prove sub cases of case 2.2, we want to prove 22 and 23.

By the correspondance lemma (CL.1), since $L_i$ is in the $i_{th}$ run of $P$ and not the first halted state in the run, $I_i = I_{A_i} \sim I_{D_i}$.

Since $I_{D_i} = \langle \rangle$ we have that $I_i = I_{A_i}$. \hfill (20)

By 20 and $|I_i| > 0$ we have $|I_{A_i}| > 0$. \hfill (21)

By 20 $I_i$ is the incremental query of $L_i$. \hfill (22)

Let $r$ be the number of solution requests for $I_i$ in $\langle C_h, ..., C_i \rangle$ where $h$ is the position of the related increment query command for $C_i$.

Let $k = |I_i|$ for some $k > 0$.

By the correspondance lemma (CL.5) and because $L_i$ is not the first IQ state of $i_{th}$ run we have $N_i[k] = r$. \hfill (23)

Let $m$ be the number of solutions for incremental query $I_i$.

Case 2.2.1: $I_i$ has at least $r$ solutions.

$I_i$ has at least $r$ solutions means that $m \geq r$. \hfill (24)

$L_i$ is the last IQ state of $i_{th}$ run and because of 24 By the correspondance lemma (CL.5) we have $r \leq |R_i[k]| \leq m$. \hfill (25)

By 23 and 25 we have $|R_i[k]| \geq N_i[k]$. \hfill (26)

By 26, 20, $C_i = next()$ and from the definition of IQ Solution defined by $P$, $S_i$ is the annotated solution derived from $R_i[k][N_i[k]]$. \hfill (27)

By 27 and 23 we have $S_i$ is the annotated solution derived from $R_i[k][r]$. \hfill (28)
By the correspondence (Cl.2), the annotated solution derived from $R_i[k][r]$ is the $r_{th}$ annotated solution of incremental query $I_i[1..k]$.

Since $|I_i| = k$ therefore $I_i[1..k]$ is $I_i$.

$S_i$ is the $r_{th}$ annotated solution for $I_i$.  

Case 2.2.2: $I_i$ has less than $r$ solutions

$I_i$ has at most $r$ solutions means that $m < r$.  

By the correspondence lemma (CL.5) and 30 we have $|R_i[k]| = m$.  

By 23, 31 and 30 we have $N_i[k] > |R_i[k]|$.  

by 32, $C_i \neq dec()$, 21 and From the definition of IQ Solution defined by $P$, if $C_i \neq dec()$, $|I_{A_i}| > 0$ and $|R_i[k]| < N_i[k]$ then

$S_i = fail$.  

$S_i = fail$.  

From 29 and 33 If $|I_i| > 0$, then let $h$ be the position of the related increment query command for $C_i$, let $k$ be the number of solution requests for $I_i$ in $\langle C_h, \ldots, C_i \rangle$. If $I_i$ has at least $r$ solutions then $S_i$ is the $r_{th}$ annotated solution for $I_i$, otherwise $S_i = fail$.  

Combining 19 and 34 we have that when $C_i = next()$, if $|I_i| = 0$ then $S_i = \Box$. If $|I_i| > 0$, then let $h$ be the position of the related increment query command for $C_i$, let $k$ be the number of solution requests for $I_i$ in $\langle C_h, \ldots, C_i \rangle$. If $I_i$ has at least $r$ solutions then $S_i$ is the $r_{th}$ annotated solution for $I_i$, otherwise $S_i = fail$.  

Case 3: $C_i = dec()$

From the definition of IQ Solution defined by $P$, If $C_i = dec()$, then $S_i = \Box$.  

147
when $C_i = \text{dec}()$ then $S_i = \Box$ \hspace{1em} (36).

By the definition of an IQ Solution to an IQ Problem and 16, 35, 36 $S$ is an IQ solution to $\langle \Pi, C \rangle$.

\[ \Box \]

**Proof $\Rightarrow$** ( uses Deterministic Run Lemma)

We must show that if $S$ is an IQ solution to the IQ problem $\langle \Pi, C \rangle$ then there exists a halted sound path $P$ in $IQTD(\Pi)$ such that $C$ is the sequence of IQ commands defined by $P$ and $S$ is the IQ solution defined by $P$. 48

Assume $S$ is an IQ solution to an IQ problem $\langle \Pi, C \rangle$. \hspace{1em} (37)

By the definition of an IQ problem, $C$ is a sequence of IQ commands of the form $\langle C_1, \ldots, C_n \rangle$

Let $I = \langle I_0, I_1, \ldots, I_n \rangle$ be the IQ sequence defined by $C$.

From the definition of $S$ being an IQ solution to $\langle \Pi, C \rangle$, $S = \langle S_1, \ldots, S_n \rangle$ we have that for $i \in [1..n]$, $S_i$ satisfies the following properties: \hspace{1em} (38)

- when $C_i = \text{inc}(Q)$, if no solution exists for $I_i$, then $S_i = \text{fail}$, otherwise $S_i$ is the first annotated solution for $I_i$.
- when $C_i = \text{next}()$, if $|I_i| = 0$ then $S_i = \Box$. If $|I_i| > 0$, then let $h$ be the position of the related increment query command for $C_i$, let $k$ be the number of solution requests for $I_i$ in $\langle C_h, \ldots, C_i \rangle$. If $I_i$ has at least $k$ solutions then $S_i$ is the $k_{th}$ annotated solution for $I_i$, otherwise $S_i = \text{fail}$.
- when $C_i = \text{dec}()$ then $S_i = \Box$.

We construct a path $P$ in $IQTD(\Pi)$ as follows:
Let $P' = \langle L_0 \rangle$ such that $L_0$ is the initial halted IQ state.

Repeat for $i \in [1..n]$ in order

- Let $L_{i-1}$ be the last state in $P'$.
- By construction $L_{i-1}$ is a halted IQ state.
- Let $\langle L_{i-1}, C_i, s', \ldots, L_i \rangle$ be the unique run as determined by the deterministic run lemma.
- $P' := P' \sim \langle L_{i-1}, C_i, s', \ldots, L_i \rangle$

$P = P'$ at the end of the above construction.

$P$ is a halted sound path in $IQTD(\Pi)$ (39)

Let $C'$ be the sequence of IQ Commands defined by $P$.

By construction of $P$, $C = C'$

$C$ is the sequence of IQ commands defined by $P$. (40)

Let $S' = \langle S'_1, \ldots, S'_n \rangle$ be the IQ solution defined by $P$.

By the proof of the theorem in the reverse direction, $S'$ is the IQ solution of the IQ problem $\langle \Pi, C \rangle$.

We show 47 holds by proving $S = S'$.

Let $L = \langle L_0, L_1, \ldots, L_n \rangle$ be the halted IQ states along $P$ where $L_0$ is the initial IQ state and for $i \in [0..n]$ $L_i$ is of the form $\langle \langle I_{A_i}, \rangle \rangle, R_i, N_i, \square, \langle \square | \square \rangle, halt \rangle$.

For $i \in [1..n]$ we consider the definitions of $S_i$ and $S'_i$ and the $i_{th}$ run of $P$, $P_i = \langle L_{i-1}, e_{i,1} = C_i, s_{i,1}, e_{i,2}, s_{i,2}, \ldots, e_{i,k_i}, L_i \rangle$.

Case 1: $C_i = dec()$:
From the definition of $S$ being the IQ solution to the IQ problem $\langle \Pi, C \rangle$, when $C_i = \text{dec}(\cdot)$, $S_i = \square$.

From the definition of $S'$ being the IQ solution defined by $P$, when $C_i = \text{dec}(\cdot)$, when $C_i = \text{dec}(\cdot)$, $S'_i = \square$.

$S_i = S'_i$. \hspace{1cm} (41)

Let $q_i = |I_i|$ and $m_i$ be the number of solutions to $I_i$.

By the correspondence lemma (CL.1) and $L_i$ not being the first IQ state in $P_i$, $I_i = I_{A_i} \sim I_{D_i}$.

From $I_{D_i} = \langle \rangle$ we have that $I_i = I_{A_i}$ and $|I_{A_i}| = q_i$.

Case 2: $C_i = \text{inc}(Q)$:

From the definition of $S$ being the IQ solution to the IQ problem $\langle \Pi, C \rangle$, when $C_i = \text{inc}(Q)$, if no solution exists for $I_i$, then $S_i = \text{fail}$, otherwise $S_i$ is the first annotated solution for $I_i$.

We consider two cases for $m_i$.

Case 2.1: $m_i = 0$

$S_i = \text{fail}$

By the correspondence lemma (CL.4), since $L_i$ is not the first IQ state of $P_i$ and $m_i = 0$ then $N_i[q_i] = 1$ and $|R_i[q_i]| = 0$.

$|R_i[q_i]| < N_i[q_i]$

By the definition of $S'$ being the IQ solution defined by $P$, If $C_i \neq \text{dec}(\cdot)$, $|I_{A_i}| = q_i > 0$ and $|R_i[q_i]| < N_i[q_i]$, then $S'_i = \text{fail}$.

$S'_i = \text{fail}$.

$S_i = S'_i$. \hspace{1cm} (42)
Case 2.2: \( m_i > 0 \)

\( S_i \) is the first annotated solution to \( I_i \).

By the correspondence lemma (CL.4), since \( L_i \) is not the first IQ state of \( P_i \) and \( m_i = 0 \) then \( N_i[q_i] = 1 \) and \( |R_i[q_i]| = 1 \).

\[ |R_i[q_i]| \geq N_i[q_i] \]

By the definition of \( S' \) being the IQ solution defined by \( P_i \), if \( C_i \neq \text{dec()} \), \( |I_{A_i}| = q_i > 0 \) and \( |R_i[q_i]| \geq N_i[q_i] \), then \( S_i' \) is the annotated solution derived from \( R_i[q_i][N_i[q_i] = 1] \).

By the correspondence lemma (CL.2), the annotated solution derived from \( R_i[q_i][1] \) is the first annotated solution of incremental query \( I_i[1..q_i] \).

\[ I_i = I_i[1..q_i] \]

\( S_i' \) is the first annotated solution of \( I_i \).

\[ S_i = S_i' \quad (43) \]

Case 3: \( C_i = \text{next()} \)

let \( h \) be the position of the related increment query command for \( C_i \) and let \( k \) be the number of solution requests or \( I_i \) in \( \langle C_h, \ldots, C_i \rangle \).

From the definition of \( S \) being the IQ solution to the IQ problem \( \langle \Pi, C \rangle \), when \( C_i = \text{next()} \), if \( |I_i| = 0 \) then \( S_i = \square \), otherwise If \( |I_i| > 0 \) and \( I_i \) has at least \( k \) solutions then \( S_i \) is the \( k_{th} \) annotated solution for \( I_i \), otherwise \( S_i = \text{fail} \).

We consider the values for \( q_i, m_i \) and \( k \).

Case 3.1: \( q_i = 0 \)

\[ S_i = \square \]

\[ |I_{A_i}| = |I_i| = q_i = 0 \]
By the definition of $S'$ being the IQ solution defined by $P$, if $C_i = \text{next}()$ and $|I_{A_i}| = 0$, then $S'_i = \square$.

$S'_{i} = \square \text{ cl.7}$

$S_{i} = S'_{i}$ (44)

Case 3.2: $q_i > 0$ and $m_i < k$

$S_i = fail$

By the correspondence lemma (CL.5), since $L_i$ is the last IQ state of $P_i$ and $m_i < k$, then $N_i[q_i] = k$ and $|R_i[q_i]| = m_i$.

$|R_i[q_i]| < N_i[q_i]$.

By the definition of $S'$ being the IQ solution defined by $P$, If $C_i \neq \text{dec}()$, $|I_{A_i}| = q_i > 0$ and $|R_i[q_i]| < N_i[q_i]$, then $S'_i = fail$.

$S'_{i} = fail$.

$S_{i} = S'_{i}$. (45)

Case 3.3: $q_i > 0$ and $m_i \geq k$

$S_i$ is the $k_{th}$ annotated solution for $I_i$.

By the correspondence lemma (CL.5), since $L_i$ is the last IQ state of $P_i$ and $m_i \geq k$, then $N_i[q_i] = k$ and $k \leq |R_i[q_i]| \leq m_i$.

$|R_i[q_i]| \geq N_i[q_i]$.

By the definition of $S'$ being the IQ solution defined by $P$, If $C_i \neq \text{dec}()$, $|I_{A_i}| = q_i > 0$ and $|R_i[q_i]| \geq N_i[q_i]$, then $S'_i$ is the annotated solution derived from $R_i[q_i][N_i[q_i] = k]$.

By the correspondence lemma (CL.2), the annotated solution derived from $R_i[q_i][k]$ is the $k_{th}$ annotated solution of incremental query $I_i[1..q_i]$.

$I_i = I_i[1..q_i]$

$S'_i$ is the $k_{th}$ annotated solution of $I_i$.  

152
\[ S_i = S'_i. \quad (46) \]

For \( i \in [1..n] \), in all cases 41, 42, 43, 44, 45, and 46 \( S_i = S'_i \).

\[ S = S'. \]

From \( S' \) being the IQ solution defined by \( P \) and \( S = S' \), \( S \) is the IQ solution defined by \( P \). \quad (47)

From showing 39, 40, and 47 under the assumption of 37 we conclude that:

If \( S \) is an IQ solution to the IQ problem \( \langle \Pi, C \rangle \) then there exists a halted sound path \( P \) in \( IQTD(\Pi) \) such that \( C \) is the sequence of IQ commands defined by \( P \) and \( S \) is the IQ solution defined by \( P \). \quad (48)