Crowd Task Language Report

Yixuan Luo

Contents

1 Introduction **1**

1.1 Program Structure 1

1.2 Notation 2

2 Language Model **3**

2.1 Sensing Model 3

2.1.1 Task Constraint 3

2.1.2 Hyper Participant 7

2.1.3 Grammar 11

2.2 Computing Model 11

2.2.1 Basic 12

2.2.2 Data Transfer 14

2.2.3 Grammar 14

2.3 CTL file 15

3 Sensory Turing Machine **17**

3.1 Automata Theory 17

3.1.1 Combinational Logic 17

3.1.2 Finite-state Machine 18

3.1.3 Push-down Automaton 18

3.1.4 Turing Machine 19

3.1.5 Sensory Turing Machine 19

3.2 Mathematic Model 19

3.3 Proofs 21
3.3.1 Proof of Power 21
3.3.2 Proof of Degradation 22
3.3.3 The Practical Value of Degradation Proofs 25

4 CTL Virtual Machine **26**

A CTL Grammar **27**
A.1 Complete Grammar 27
A.2 Terminator Characteristics 29
A.2.1 Identifier 29
A.2.2 TimeLiteral 29
A.2.3 DateLiteral 29
A.2.4 Coord 30
A.2.5 Numeric Literals 30
A.2.6 Indent Related 30

B Example Applications **31**
B.1 Video Doc 31
B.2 Journalist 32
B.3 Map3D 33
B.4 Broken Streetlight 33
B.5 Missing Manhole 34
B.6 Rebalance Bicycle 35
B.7 Busy Canteen 37
B.8 Noise Detection 37

Chapter 1

Introduction

CTL (Crowd Task Language) is a language for the field of Crowdsourcing and Crowdsensing, which provides the ability to describe the data collection process at a granular level, as well as the ability to describe the computation in a way that is equivalent to the capabilities of other high-level programming languages (C, Python, Java, etc.).

Other high-level programming languages solve problems in such a way that the language is concerned with the computational process of the data after it has been collected. But CTL wants to extend this paradigm, where the programming language's capabilities will be extended to before data collection. The principle behind this is based on an extended Turing machine theory called Sensing Turing Machine Theory (which will be introduced in chapter 3).

This report defines the syntax for CTL programs and an informal abstract semantics for the meaning of such programs.

1.1 Program Structure

A CTL program can be viewed as two parts, the part describing data sensing and the part describing data computation. The collection process is described using the following concepts. The collection process is described using the following concepts:

- Task Constraints
- Hyper Participants

The task constraints determine the structure of the sensing task itself, and the hyper participants describe the specific data types required for the task.

In the computational part of the CTL, a hybrid object-oriented and functional programming paradigm is provided. It has the following features:

- Object-oriented
- high-order function
- list, tuple and dict comprehensions
- a rich set of primitive datatype

1.2 Notation

The descriptions of lexical analysis and syntax use a mixture of EBNF and PEG. This uses the following style of definition:

$$taskUnit \rightarrow \texttt{'CrowdTask'}\ \texttt{Identifier}\ \texttt{'\{'}\ main\ \texttt{'\}'}$$

Each rule begins with a name (which is the name defined by the rule) and →. A vertical bar (|) is used to separate alternatives; it is the least binding operator in this notation. A star (∗) means zero or more repetitions of the preceding item; likewise, a plus (+) means one or more repetitions, and a phrase enclosed in square brackets ([]) means zero or one occurrence (in other words, the enclosed phrase is optional). The * and + operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

Chapter 2

Language Model

The CTDL language model is divided into two parts: the sensing model and the computing model. Mathematical theory of language in chapter 3.

2.1 Sensing Model

The CTL sensing model describes the data collection process using Task Constraints and Hyper Participant. The mathematical theory behind the sensing model is the sensing capability of the sensing Turing machine, see Chapter 3.

2.1.1 Task Constraint

Task constraints are constraints that describe a crowd task itself, e.g., time constraints, spatial constraints, and dependence constraints. The task constraints are not directly related to the participants performing the task, but can directly affect the task scheduling of the system. Listing 2.1 shows a complete code snippet of the constraint definition.

Listing 2.1: CTL Constraints Part Snippet

```
Constraints:
    Temporal:
        10:00 - 12:00
        16:00 - 18:00

    Date:
        2023/2/10 - 2023/2/28

```

```
    Spatial:
        (108.93,34.32) - (110.93,54.32)

    Dependence:
        Task1 -> Task2 -> Task3
        Task3 -> Task4

    dataQuantity: 100

    Repeat: INF

    Priority: 5
```

There are two types of 'dimension' concepts, namely task dimension and constraint dimension. The task dimension affects the scheduling of tasks, and the constraint dimension is used to describe a class of constraints. Each constraint dimension belongs to a task dimension. The currently supported task dimensions and constraint dimensions are shown in Table 2.1.

Task Dim.	Constraint Dim.
Time Dimension	Temporal Date
Space Dimension	Spatial
Precedence Dimension	Dependence Priority
Quantity Dimension	Repeat DataQuatity

Table 2.1: Task Dimensions

Temporal

The Temporal dimension supports the use of a 24-hour (or 12-hour) system to describe the time that a task is executed with a minimum precision of 1 minute (as listing 2.2 shows). The specific constraint types that can be used are:

- Temporal Interval: `TemporalInterval(10:00, 12:00)` or its syntactic sugar version: `10:00 - 12:00`.
- Temporal Point: `TemporalPoint(15:30)` or its syntactic sugar version: `15:30`.
- Expression, which results in Temporal.

Listing 2.2: Temporal Dimension Snippet

```
Temporal:
    TemporalInterval(10:00, 12:00)
    10:00 - 12:00
    Point(15:31)
    15:31
    Task1.returnData().Temporal()
```

Date

Date constraints provide a coarse-grained description of time. Date literals can be used to describe the start time and deadline of a task. The constraint types that can be used include:

- FromNowTo: `FromNowTo(2023/2/13)`.
- DateInterval(2023/2/10, 2023/2/28) and its syntactic sugar version: `2023/2/10 - 2023/2/28`

Listing 2.3: Date Dimension Snippet

```
Date:
    2023/2/10 - 2023/2/28
    FromNowTo(2023/2/12)
    DateInterval(2023/2/10, 2023/2/28)
```

Spatial

The Spatial dimension supports 2-dimensional spatial regions described using latitude and longitude (as listing 2.4 shows). Specific types include:

- Point Location: `PointLocation(108.93, 43.32)` or its syntactic sugar version: (108.93, 43.32).
- Line: `Line([(108.93,34.32), (110.93,54.32), (138.93,14.32)])`.
- Polyon: `Polygon([(108.93,34.32), (110.93,54.32), (138.93,14.32)])`
- Rectangle region constraint declares syntactic sugar: `(108.93,34.32) - (110.93,54.32)`.
- Expression, which results in Spatial.

Listing 2.4: Spatial Dimension Snippet

```
Spatial:
    PointLocation (108.93, 34.32)
    (108.93,34.32)
    Line([
        (108.93,34.32),
        (110.93,54.32),
        (138.93,14.32)
        ])
    Polygon([
        (108.93,34.32),
        (110.93,54.32),
        (138.93,14.32)
        ])
    (108.93,34.32) - (110.93,54.32)
    Task2.Location
```

Dependence

Dependence constraints support defining dependency concerns between multiple tasks. All dependencies need to form partial order relationships. CTL provides a `->` operator to define task dependencies, as shown in 2.5.

Listing 2.5: Dependence Dimension Snippet

```
Dependence:
    Task1 -> Task2 -> Task3
    Task3 -> Task4
```

The `"PreTask->SuccTask"` statement implies that the PreTask must have been executed before the SuccTask is executed. At the same time, SuccTask can access the results of PreTask.

Priority

The Priority constraint uses an integer to indicate the absolute priority of the task. The optional range is 1-9, the higher the number, the higher the priority. The default value is 5.

Repeat

The Repeat constraint defines the number of repeated executions of the task. An integer can be used to specify the exact number of repeated executions, or the keyword INF can be used to indicate that the task will be repeated infinitely until

the task is manually canceled, as shown in listing 2.6. It should be noted that it cannot be used with the Date dimension at the same time.

Listing 2.6: Repeat Dimension Snippet

```
Repeat:
    INF
    10
```

Combination Constraint Types

Instances of different constraint types can be combined under the same constraint dimension. In this case, the instances of each constraint type are logically related to each other. For example, listing 2.7.

Listing 2.7: Combination Constraint Type Snippet

```
Spatial:
    Polygon([
        (108.93,34.32),
        (110.93,54.32),
        (138.93,14.32)
        ])
    (108.93,34.32) - (110.93,54.32)
```

2.1.2 Hyper Participant

Crowd Intelligence tasks can and can only be completed by numerous participants, and the task requires many capabilities from the participants, such as some perceptual capability or some computational capability, or even some intelligence.

In CTL, the developer of a Crowd Intelligence task only needs to consider what data and abilties the task participants provide, and CVM will break down the hyper-participants into real participants in the system in a reasonably efficient way and hand them over to CrowdOS for processing. A code snippet of the hyperparticipant declaration is shown in 2.8.

Listing 2.8: Hyper Participant Snippet

```
Hparticipant:
    Location
    nullable LocalTime
    Image
    HumanIntelligence("take a picture")
    Specific("skill:RoadRepair")
```

Ability

An ability is the smallest atomic characteristic of a hyperparticipant. An ability either specifies a characteristic that the hyperparticipant needs to have or specifies the type of data that the hyperparticipant needs to return, where the latter ability is called **returnable**.

A ability can be declared using a statement like `Image()`. For some capabilities (e.g., HumanIntelligent and Specific), some optional constructor parameters can be accepted, and information about these parameters is passed to CVM and CrowdOS. For capabilities that do not require constructor parameters, `()` can be omitted. That is, `Image` is equivalent to `Iamge()`. For example listing 2.9.

Listing 2.9: Ability Create Snippet

```
1 Hparticipant:
2     Image()  # Equivalent to Image
3     LocalTime  # Equivalent to LocalTime()
```

The currently supported capabilities and their meanings are shown in table 2.2.

Ability	Implication	Returnable	Return Type	Args	Arg Behavior
HumanIntelligence	Requires participants to have a certain level of intelligence	×	None	String	Passes the specified text message to the participant with intelligence.
Specific	Specify a specific type of participant in CrowdOS	×	None	String	Specifies a special type of participant in the form of a String. See the CVM section and the CrowdOS system interface for details.
Location	Require participants to be able to provide location data	√	Location	/	/
LocalTime	Require participant to return local time	√	Time	/	/
Text	Require participant to return text data	√	String	/	/
Video	Require participant to return video data	√	Video	/	/
Image	Require participant to return image date	√	Image	/	/
Acceleromenter	Require participant to return acceleration sensor data	√	Acceleromenter	/	/
Gyroscope	Require participant to return gyroscope data	√	Gyroscope	/	/
Light	Require participant to return light sensor data	√	LightData	/	/
Magnetic_field	Require participant to return magnetic field sensor data	√	MagneticField	/	/
Orientation	Require participant to return orientation sensor data	√	Orientation	/	/
Pressure	Require participant to return pressure sensor data	√	Pressure	/	/
Proximity	Require participant to return proximity sensor data	√	Proximity	/	/
Temperature	Require participant to return temperature sensor data	√	Temperature	/	/

Table 2.2: Hparticipant Abilties

Data Return Behavior

Hparticipant defines all the data types required by a CrowdTask, as well as the form of the data tuples to be passed to the CTL calculation section. For example, in listing 2.10, six capabilities are defined that will be passed to the computation part in the form of tuples of 2.1 when a data collection is completed.

Listing 2.10: Data Return Snippet

```
1 Hparticipant:
2     Location
3     LocalTime
4     Image
5     Text
6     Specific("skill:RoadRepair")
7     HumanIntelligence("Describe what you see")
```

```
(Location, Time, Image, Text)                    (2.1)
```

In tuple 2.1, the four dimensions are the data returned by the returnable capability declared in Hparticpant. The last two ability in Listing 2.10 is not returnable, so it does not appear in the data tuple.

Hparticipant enables CTL programs to collect data using a virtual powerful participant. In practice, a single piece of data may be collected by multiple participants. For example, in 2.1, `(Location, Image)` are provided by real participant 1, and `(Time, Text)` are provided by real participant 2. A single piece of data provided by several participants is usually related to the CrowdOS system state and is unpredictable.

`nullable` Keyword

A capability may be declared nullable. If a capability is declared nullable, the corresponding dimension in the returned data meta ancestor may be None. e.g., in Listing 2.8.

The CTL computation part receives a data tuple in two possible cases, as shown in 2.2 and 2.3. If all capabilities are not nullable, the computation is performed only when all dimensional data are collected. When nullable capabilities exist in the Hparticipant, the timing of the entry data return is determined by the CVM.

```
(Location, Time, Image)                          (2.2)
```

```
(Location, None, Image)                          (2.3)
```

2.1.3 Grammar

The grammar of Task Constraints is as following:

```
constraintDef → 'Constraints' ':' NEWLINE INDENT cDim+ DEDENT
        cDim → constraintDim ':' cSuite
      cSuite → (cStmt | NEWLINE INDENT cStmt+ DEDENT)
constraintDim → 'Temporal' | 'Spatial' | 'Precedence' | 'Date'
              | 'Priority'
      cStmt → expression (';')? NEWLINE
 expression → primary
              | expression bop='.' (Identifier | callOrCreate)
              | expression '[' expression ']'
              | callOrCreate
              | expression bop=('*' | '/' | '%') expression
              | expression bop=('+' | '-') expression
              | <assoc=right> Identifier bop='->' Identifier
    primary → '(' (listComp)? ')'
              | '[' (listComp)? ']'
              | Identifier
              | literal
   listComp → expression (',' expression)* (',')?
    literal → integerLiteral
              | 'INF'
              | floatLiteral
              | TimeLiteral
              | DateLiteral
integerLiteral → (DecimalLiteral | HexLiteral | OctLiteral)
 floatLiteral → (FloatLiteral | HexFloatLiteral)
callOrCreate → Identifier ('(' expressionList? ')')?
expressionList → expression (',' expression)*

    HpartDef → 'Hparticipant' ':' hSuite
      hSuite → NEWLINE INDENT abilityDecl+DEDENT
abilityDecl → ('nullable')? callOrCreate
```

2.2 Computing Model

CTL's computational model is a high-level programming language that incorporates object-oriented and functional programming.

2.2.1 Basic

The characteristics underlying the CTL computational model are as follows.

Expressions

Expressions are computable statement:

```
1 + 1
```

You can output the results of expressions using `CrowdOS.println`:

```
CrowdOS.println(1)  # 1
CrowdOS.println(1 + 1)  # 2
CrowdOS.println("Hello!")  # Hello!
CrowdOS.println("Hello," + " world!")  # Hello, world!
```

Variable

You can define a variable with the `var` keyword, and you can re-assign them.

```
var x = 1 + 1
x = 3
CrowdOS.println(x * x)  # 9
```

As with values, the type of variable can be omitted and inferred, or it can be explicitly stated:

```
var x: Int = 1 + 1
```

Blocks

You can combine expressions by surrounding them with `{}`. We call this a block. The result of the last expression in the block is the result of the overall block, too:

```
CrowdOS.println({
  val x = 1 + 1
  x + 1
}) // 3
```

Function

Function are defined with the `compute` keyword. `compute` is followed by a name, parameter list(s), a return type, and a body:

```
compute add(x: Int, y: Int): Int = x + y
CrowdOS.println(add(1, 2)) // 3
```

A Function can take multiple parameter lists:

```
compute addThenMultiply(x: Int, y: Int)(multiplier: Int): Int
    = (x + y) * multiplier
CrowdOS.println(addThenMultiply(1, 2)(3)) // 9
```

Or no parameter lists at all:

```
compute name: String = CrowdOS.getProperty("host")
CrowdOS.println("Hello, " + name + "!")
```

Methods can have multi-line expressions as well:

```
def getSquareString(input: Double): String = {
  var square = input * input
  square.toString
}
CrowdOS.println(getSquareString(2.5)) // 6.25
```

Classes

You can define classes with the class keyword, followed by its name and constructor parameters:

```
class Greeter(prefix: String, suffix: String) {
  def greet(name: String): Void =
    CrowdOS.println(prefix + name + suffix)
}
```

In class you can use `def` keyword to define a method. The other feature of method is same to function. You can use the class name followed parameters list to create an instance:

```
var greeter = Greeter("Hello, ", "!")
greeter.greet("CTL developer") // Hello, CTL developer!
```

2.2.2 Data Transfer

The purpose of the CTL computational model is to compute the data passed from the sensing model. The data is passed through a special object Hpart, and in the function, the for data can be obtained using Hpart.dataName, where the data name is the same name as the ability in the Hparticipant. For example, when the Hparticipant is declared as listing 2.11, the following operation gets the data of interest.

Listing 2.11: Data Transfer Snippet

```
1  Hparticipant:
2      Location
3      nullable LocalTime
4      Image
5      HumanIntelligence("take a picture")
6      Specific("skill:RoadRepair")
7
8
9  compute main(): Void = {
10     CrowdOS.println(Hpart.Loaction)
11     CrowdOS.println(Hpart.LocalTime)
12     CrowdOS.println(Hpart.Image)
13     CorwdOS.println(Hpart.Pid)
14 }
```

For each returned capability in Hparticipant, there is a property with the same name in the Hpart object. In addition, Hpart provides some inspection capabilities, for example, for line 13 in listing 2.11, Hpart.pid returns a list of real participants who have contributed to that data.

2.2.3 Grammar

The grammar of class and Function are as follow:

$$
\begin{aligned}
\mathit{topDef} &\rightarrow \texttt{'class'}\ \mathit{classDef}\\
\mathit{classDef} &\rightarrow \texttt{Identifier}\ \mathit{classParamClauses}\ \mathit{classBody}\\
\mathit{classParamClauses} &\rightarrow (\texttt{'('}\ \mathit{classParams}\ \texttt{')'})*\\
\mathit{classParams} &\rightarrow \mathit{classParam}\ (\texttt{','}\ \mathit{classParam})*\\
\mathit{classParam} &\rightarrow (\texttt{'val'} \mid \texttt{'var'})?\ \texttt{Identifier}\ \texttt{':'}\ \texttt{Identifier}\ (\texttt{'='}\ \mathit{expression})?\\
\mathit{classBody} &\rightarrow \texttt{'\{'}\ \mathit{classBodyDecl}\texttt{+}\ \texttt{'\}'}\\
\mathit{classBodyDecl} &\rightarrow (\mathit{filedDecl} \mid \mathit{methodDecl})\\
\mathit{filedDecl} &\rightarrow \texttt{'var'}\ \texttt{Identifier}\ \texttt{':'}\ \texttt{Identifier}\ (\texttt{'='}\ \mathit{expression})?\\
\mathit{methodDecl} &\rightarrow \texttt{'def'}\ \mathit{procedure}\\
\mathit{procedure} &\rightarrow \mathit{signature}\ (\texttt{':'}\ \texttt{Identifier})?\ \texttt{'='}\ \mathit{procBody}\\
\mathit{procBody} &\rightarrow (\mathit{expression} \mid \mathit{block})
\end{aligned}
$$

$$
\begin{aligned}
\textit{block} &\rightarrow \texttt{'\{'}\ \textit{blockStm}*\ \texttt{'\}'}\\
\textit{signature} &\rightarrow \texttt{Identifier}\ (\texttt{'('}\ \textit{params}?\ \texttt{')'})*\\
\textit{params} &\rightarrow \textit{param}\ (\texttt{','}\ \textit{param})*\\
\textit{param} &\rightarrow \texttt{Identifier}\ (\texttt{':'}\ \texttt{Identifier})?\ (\texttt{'='}\textit{expression})?\\
\textit{blockStm} &\rightarrow \textit{localVarDecl}\ \texttt{NEWLINE}\\
&\quad |\ \textit{statement}\ \texttt{NEWLINE}\\
\textit{localVarDecl} &\rightarrow \texttt{'var'}\ \texttt{Identifier}\ (\texttt{'='}\ \textit{expression})?\\
\textit{statement} &\rightarrow \textit{block}\\
&\quad |\ \texttt{'if'}\ \texttt{'('}\ \textit{expression}\ \texttt{')'}\ \textit{statement}\ (\texttt{'else'}\ \textit{statement})?\\
&\quad |\ \texttt{'break'}\ \texttt{NEWLINE}\\
&\quad |\ \texttt{'continue'}\ \texttt{NEWLINE}\\
&\quad |\ \textit{expression}\ \texttt{NEWLINE}\\
\\
\textit{funcDecl} &\rightarrow \texttt{'compute'}\ \textit{procedure}
\end{aligned}
$$

2.3 CTL file

A CTL file consists of multiple CrowdTasks and Classes. A CrowdTask consists of Constraints, Hparticipant and functions used for computation. As shown in listing 2.12.

Listing 2.12: CTL file

```
1  CrowdTask BrokenStreetlight {
2      Constraints:
3          Precedence:
4              BrokenStreetlightSense -> BrokenStreetlightAction
5              BrokenStreetlightAction -> BrokenStreetlightReview
6
7      CrowdTask BrokenStreetlightSense {
8          Constraints:
9              Spatial:
10                 (108.80, 34.33) - (109.05, 34.20)
11
12         Hparticipant:
13             Location
14             image
15
16         compute main(): BrokenStreetlightInfo = {
17             BrokenStreetlightInfo(Hpart.Location, Hpart.Image)
18         }
19     }
20
21 }
22
23 CrowdTask BrokenStreetlightAction {
24     Constraints:
25         Spatial:
```

```
            BrokenStreetlightSense.location
        Priority: 9

    Hparticipant:
        Specific("skill:PowerLineRepair")

    compute main() = {
        CrowdOS.save("./data/BrokenStreetlightLog/")
    }
}

CrowdTask BrokenStreetlightReview {
    Constraints:
        Spatial:
            BrokenStreetlightSense.Location


    Hparticipant:
        Location
        Image

    compute main(): Void = {
        CrowdOS.save(
            "./data/BrokenStreetlightReview/",
            (Hpart.Pid, Hpart.Location, Hpart.Image)
            )
    }
}

class BrokenStreetlightInfo(
    var location: Location,
    var img: Image
){
    def getLocation() = location
    def getImg() = img
}
```

The grammar of CTL file are as following:

$$
\begin{aligned}
ctlFile &\rightarrow (topDef)* \\
topDef &\rightarrow \texttt{'CrowdTask'}\ taskDef \\
&\quad \mid \texttt{'class'}\ classDef \\
taskDef &\rightarrow (constraintDef \mid HpartDef \mid funcDecl)*
\end{aligned}
$$

Chapter 3

Sensory Turing Machine

The theory behind the CTL language is sensory Turing machine theory. Sensory Turing machines belong to a theory of automata. This chapter first briefly introduces other theories in automata theory, then describes the mathematical model of sensory Turing machines in detail, and finally does some important proof work based on sensory Turing machine theory.

3.1 Automata Theory

In this section, we present each theory in classical automata theory in turn, from simple to complex, with the emphasis on reflecting, in as uniform a mathematical form as possible, how the latter theory improves in power compared to the former one. Eventually, the place of sensory Turing machines in the overall theory of automata is given, and the details of the mathematical model are discussed in the next section.

3.1.1 Combinational Logic

In automata theory, combinational logic is a type of digital logic which is implemented by Boolean circuits, where the output is a pure function of the present input only.

The mathematical form of combinatorial logic is a binary group (eq. 3.1).

$$(\Sigma, \delta) \tag{3.1}$$

where

- $\Sigma = 1, 0$ is the input alphabet. 0 and 1 means true and false.
- $\delta : \Sigma \rightarrow \Sigma$ is pure function.

3.1.2 Finite-state Machine

Finite-state machine is an abstract machine that can be in exactly one of a finite number of states at any given time.

The mathematical form of Finite-state machine is a five tuple (eq. 3.2)

$$(Q, \Sigma, q_0, \delta, F) \tag{3.2}$$

where

- Q is a finite set of states.
- Σ is a finite set of input alphabets.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transfer function.
- q_0 is the initial state, $q_0 \in Q$ or $q_0 \subset Q$
- $F \subset Q$ is the set of final states.

3.1.3 Push-down Automaton

A pushdown automaton (PDA) is a type of automaton that employs a stack.

The mathematical form of pushdown automaton is a six tuple (eq. 3.3)

$$(Q, \Sigma, \Gamma, \delta, q_0, F) \tag{3.3}$$

where

- Q is a finite set of states.
- Σ is a finite set of input alphabets.
- Γ is a finite set of stack alphabets.
- $\delta : Q \times \Gamma_\epsilon \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transfer function
- q_0 is the initial state, $q_0 \in Q$ or $q_0 \subset Q$
- $F \subset Q$ is the set of final states.

3.1.4 Turing Machine

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.

The mathematical form of Turing machine is a six tuple (eq. 3.4)

$$(Q, \Sigma, \Gamma, \delta, q_0, F) \tag{3.4}$$

where

- Q is a finite set of states.
- Σ is a finite set of input alphabets.
- Γ is a finite set of stack alphabets.
- $\delta : Q \times \Gamma \rightarrow (Q \cup F) \times \Gamma \times \{R, L, -\}$
- q_0 is the initial state, $q_0 \in Q$ or $q_0 \subset Q$
- $F = \{q_{accept}, q_{reject}\}$ is the set of final states.
- $q_{accept} \in Q$ is the accept state,
- $q_{reject} \in Q$ is the reject state.

3.1.5 Sensory Turing Machine

3.2 Mathematic Model

A sensory Turing machine is an extension of a Turing Machine that extends a multi-tape Turing Machine (actually a 2-tape Turing Machine) with the ability to perceive external objects. Its mathematical model is a six-tuple.

$$ST(Q, \sigma, \Gamma, \delta, q_0, F) \tag{3.5}$$

where

- Q is a non-empty set of infinite states, Q does not contain the stop state F.
- Σ is a non-empty infinite input alphabet that does not contain the blank character Δ.

- Γ is a non-empty infinite tape alphabet containing blank characters $\Delta, \Sigma \subseteq \Gamma$.
- $\delta : Q \times \Gamma^2 \rightarrow (Q \cup F) \times \Gamma^2 \times \{P, OB\}^2 \times \{L, R, S\}^2$ is the transfer function, P is the sensory Turing machine printing paper tape operation with the same capability as the Turing machine, $PS(S \in \Gamma)$ means printing symbol S in the square where the current read/write head is located. OB is the operation of the sensory Turing machine sensing external objects, $OBS(S \in \Gamma)$ means sending symbol S to the perceptible external object sends the symbol S. The external perceptible object prints the response symbol string on the second paper strip of the sensory Turing machine. In particular, $OB\Delta$ indicates that no operation is performed.
- q_0 is the initial state.
- $F = \{q_{accept}, q_{reject}\}$ is the set of downtime states.

The mathematical model of the perceptible object is a binary (eq. 3.6).

$$SO(\varrho, \Delta t) \tag{3.6}$$

where

- $\varrho : \Gamma^n \rightarrow \Omega^m$ denotes the response of a perceptible object to a given input. ϱ is a partially applied mapping of the mapping $\varrho^* : \Gamma^n \times \mathcal{R} \rightarrow \Omega^m$, i.e., $\varrho(s) = \varrho^*(s, \tau)$, $(\tau - 0) \geq \Delta t$, Γ is an infinite tape alphabet of sensory Turing machines. Ω is an infinite symbol table.
- $\Delta t \in \mathcal{R}$ is the time sensitivity of ϱ, $\mathcal{R}$ is the set of real number.

The mapping ϱ^* has the following properties.

$$\varrho^*(s_1, t_1) = \Delta \tag{3.7}$$
$$\varrho^*(s_2, t_2) = \Delta \tag{3.8}$$
$$\vdots$$
$$\varrho^*(s_n, t_n) = (\alpha_1, \alpha_2, \ldots, \alpha_n) \tag{3.9}$$

where

$$\alpha_i \in \Omega \tag{3.10}$$
$$s_i \in \Gamma \tag{3.11}$$
$$t_1 < t_2 < \cdots < t_i < \cdots < t_n \tag{3.12}$$
$$t_n - t_1 \leq \Delta t \tag{3.13}$$

This property gives ϱ a good property of having the following equation 3.14 in Δt time.

$$\varrho(s_1, s_2, \ldots, s_n) = \sum_{i=1}^{n} \varrho(e^i_{s_i}) = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \Omega \tag{3.14}$$

where, $e^i_{s_i}$ means that the ith dimension of the vector is the symbol s_i and all other dimensions are Δ. For example, $e^1_{s_1} = (s_1, \Delta, \ldots, \Delta)$, $e^n_{s_n} = (\Delta, \ldots, \Delta, s_n)$.

A sensory Turing machine and a perceptible object model form a sensory Turing machine dialogue model (eq. 3.15).

$$DIA(ST, SO) \tag{3.15}$$

It is important to note that the mathematical form of the above sensory Turing machine and perceptible object model is not unique. For example, we can remove the dimension about time from the perceptible object model, i.e., $SO(\varrho)$. Also, add another paper tape for the sensory Turing machine as an output buffer for the OB perception operation. However, such a model will lead to an overly complex sensory Turing machine, which is not conducive to our subsequent proof work.

3.3 Proofs

A sensory Turing machine is not equivalent to a Turing machine, specifically a sensory Turing machine can simulate a Turing machine, but a Turing machine cannot simulate a sensory Turing machine. The proof of inequivalence is divided into two parts, first proving that a sensory Turing machine can simulate a 2-tape Turing machine, and then proving that a Turing machine can simulate a degraded sensory Turing machine under certain qualifications.

3.3.1 Proof of Power

The idea of the power proof is to first prove that a sensory Turing machine can simulate a 2-tape Turing machine, after proving that a 2-tape Turing machine is equivalent to a general Turing machine, thus proving that a sensory Turing machine can simulate a general Turing machine.

Proof. 1. Given any 2-tape Turing machine $T(Q, \Sigma, \Gamma, \Delta, q_0, F)$, where

$$\delta : Q \times \Gamma^2 \to (Q \cup F) \times \Gamma^2 \times \{L, R, S\}^2 \tag{3.16}$$

2. Since the 2-tape Turing machine only prints P operations, δ can be rewritten as

$$\delta^* : Q \times \Gamma^2 \rightarrow (Q \cup F) \times \Gamma^2 \times \{P\}^2 \times \{L, R, S\}^2 \tag{3.17}$$

3. Let the set Ψ be the set of derivatives of δ.

$$\Psi = \{\delta^*(q_1, s_1) = (q_2, s_2, P, op) | q_1 \in Q, q_2 \in (Q \cup F), \\ s_1, s_2 \in \Gamma, op \in \{L, R, S\}\} \tag{3.18}$$

4. For any sensory Turing machine $ST(Q_{st}, \Sigma_{st}, \Gamma_{st}, \delta_{st}, q_0, F_{st})$

$$\delta_{st} : Q_{st} \times \Gamma_{st}^2 \rightarrow (Q_{st} \cup F_{st}) \times \Gamma_{st}^2 \times \{P, OB\}^2 \times \{L, R, S\}^2 \tag{3.19}$$

5. Let the set Ψ_{st} be the set of derivatives of δ_{st}.

$$\Psi_{st} = \{\delta_{st}(q_1, s_1) = (q_2, s_2, b, op) | q_1 \in Q_{st}, q_2 \in (Q_{st} \cup F_{st}), \\ b \in \{P, OB\}, s_1, s_2 \in \Gamma_{st}, op \in \{L, R, S\}\} \tag{3.20}$$

6. By 3 and 5, then $\Psi \subset \Psi_{st}$

7. For any Turing machine, there exists a sensory Turing machine

$$ST(Q, \Sigma, \Gamma, \delta, q_0, F) \tag{3.21}$$

where

$$\delta \in \Psi \subset \Psi_{st} \tag{3.22}$$

8. From 7, there exists a perceptual Turing machine that can simulate any 2-tape Turing machine.

9. It is known that a 2-tape Turing machine is equivalent to a 1-tape Turing machine. (ref todo)

10. From 8 and 9, there exists a sensory Turing machine that can simulate any Turing machine.

□

The proof 3.3.1 shows that the power of the sensory Turing machine is greater than or equal to that of the Turing machine.

3.3.2 Proof of Degradation

Sensory Turing machines can degrade into ordinary Turing machines (variants that can be simulated by Turing machines) under certain restrictions.

Trivial Degradation

In the proof of 3.3.1, a degradation of the sensory Turing machine has been embodied. When we restrict the operation of the sensory Turing machine as 3.23, and An arrow with d indicates that this is a degradation operation of the perceptual Turing machine.

$$\{P, OB\} \xrightarrow{d} \{P\} \tag{3.23}$$

At this point, for the set Ψ of derivatives of δ in a 2-tape Turing machine and the set Ψ_{st} of derivatives of the sensory Turing machine δ_{st} have

$$\Psi = \Psi_{st} \tag{3.24}$$

That is, a sensory Turing machine is equivalent to a 2-tape Turing machine under the condition of trivial degradation. That is, a Turing machine can simulate a sensory Turing machine with a trivial degradation.

Perceptible Object Degradation

In a sensory Turing machine dialogue model 3.25.

$$DIA(ST, SO) \tag{3.25}$$

We do not assume any a priori knowledge of the perceptible object model; a perceptible object can be anything that can respond to an input. From this, we can propose another qualification that there exists a class of perceptible objects $SO_M(\varrho_m, \Delta t)$ which is a Turing machine $T_{so}(Q, \Sigma_{so}, \Gamma, \delta, q_0, F)$, which we call a perceptible object Turing machine, where

$$\varrho_m : \Gamma^n \to \Gamma^n \tag{3.26}$$

$$\Sigma_{so} = \Gamma \tag{3.27}$$

In addition, we qualify the sensory Turing machine ST^*: when the sensory Turing machine performs OB operations, it is printed on the paper tape of the perceptible object Turing machine instead.

At this point the sensory Turing machine dialogue model degrade into two interacting Turing machines as 3.28.

$$DIA(ST, SO) \xrightarrow{d} DIA^*(ST^*, T_{so}) \tag{3.28}$$

The two Turing machines interact in the following form, first the degraded sensory Turing machine ST^* reads and writes symbols on the first tape, and when a

certain state transfer requires an OBs operation, ST^* prints character s on T_{so}'s paper tape. t_{so} is in a circular state (circular todo) at the initial state, and its skeleton tables (# todo ref) are as follows:

m-config.	Symbol	Operations	Final m-config.
$\mathfrak{W}$	Δ	S	$\mathfrak{W}$
$\mathfrak{W}$	not Δ	$op \in (\{P, OB\} \times \{L, R, S\})$	$\mathfrak{C}$
...	...	...	...

$\mathfrak{W}$ is called the wait state of T_{so}, once ST^* performs an OB operation, T_{so} prints the response on the second paper tape of ST^*.

The following proof shows that the degradation sensory Turing machine dialogue model $DIA^*(ST^*, T_{so})$ can be simulated by a 2-tape Turing machine T. From the trivial degradation proof (3.3.2) it follows that a trivial degraded perceptual Turing machine is equivalent to a Turing machine. Thus, T can simulate the computational process in ST that does not contain OB, and T can also simulate T_{so}. Thus, it is only necessary to construct a simulation process to simulate the OB operation of ST^* and its interaction with T_{so}.

Proof. 1. T is in the q_{st} state and prints the separator character # on the second tape first, and in one (or several) steps, prints the character. Eventually the separator character # is printed again.

$$(q_{st}, x_1\underline{a_1}y_1, \underline{\Delta}) \vdash_T^* (q'_{st}, x_1\underline{a_1}y_1, \#s\underline{\#}) \tag{3.29}$$

The formula uses the symbolic representation of Martin ([1] todo), where the triple represents a 2-tape Turing machine, the first component is the current state the Turing is in, and the second component represents the symbol on paper tape 1, where x_1,y_1,s represents the symbol string, a_1,# represents the character, and Δ represents the null character. The underscore under the character indicates the position where the Turing machine read/write head is located. $\vdash$ indicates that the previous Turing machine state can be loaded with another state by a one-step operation, and $\vdash^*$ indicates by a 0-step or multi-step operation.

2. T is transformed into q_{so} state, and the read/write head of the second paper strip looks for the first # to the left, after which it enters the calculation process and loses the calculation again after the # in the format $@r@$. That is

$$(q'_{st}, x_1\underline{a_1}y_1, \#s\underline{\#}) \vdash_T^* (q_{so}, x_1\underline{a_1}y_1, \#s\#@r\underline{@}) \tag{3.30}$$

3. T is transformed into the q''_{st} state, and the read/write head of the second paper strip looks for the first @ to the left, after which it enters the

computation process straight to the encounter @ (the second @). That is

$$(q_{so}, x_1\underline{a_1}y_1, \#s\#@r\underline{@}) \vdash_T^* (q''_{st}, x_2\underline{a_2}y_2, \#s\#@r\underline{@}) \tag{3.31}$$

4. Finally, delete all characters on the second paper strip and wait for the next OB operation.

$$(q''_{st}, x_2\underline{a_2}y_2, \#s\#@r\underline{@}) \vdash_T^* (q'''_{st}, x_2\underline{a_2}y_2, \underline{\Delta}) \tag{3.32}$$

□

3.3.3 The Practical Value of Degradation Proofs

The degradation proof of a sensory Turing machine states that we can implement a system that can simulate a sensory Turing machine with a generic Turing machine and certain artifacts (or artificial intelligence) under certain constraints. This theory is also the essence of CTL.

Chapter 4

CTL Virtual Machine

Ongoing performance trials to be published soon.

Appendix A

CTL Grammar

A.1 Complete Grammar

The grammar of CTL uses the EBNF(Extended Backus-Naur Form), where: The notation of CTDL is a mixture of EBNF and PEG.

$$
\begin{aligned}
ctlFile &\rightarrow (topDef)* \\
topDef &\rightarrow \texttt{'CrowdTask'}\ taskDef \\
&\quad | \ \texttt{'class'}\ classDef \\
taskDef &\rightarrow (constraintDef \mid HpartDef \mid funcDecl)* \\
\\
constraintDef &\rightarrow \texttt{'Constraints' ':' NEWLINE INDENT}\ cDim+\ \texttt{DEDENT} \\
cDim &\rightarrow constraintDim\ \texttt{':'}\ cSuite \\
cSuite &\rightarrow (cStmt \mid \texttt{NEWLINE INDENT}\ cStmt+\ \texttt{DEDENT}) \\
constraintDim &\rightarrow \texttt{'Temporal'} \mid \texttt{'Spatial'} \mid \texttt{'Precedence'} \mid \texttt{'Date'} \\
&\quad | \ \texttt{'Priority'} \\
cStmt &\rightarrow expression\ (\texttt{';'})?\ \texttt{NEWLINE} \\
expression &\rightarrow primary \\
&\quad | \ expression\ \texttt{bop='.'}\ (\texttt{Identifier} \mid callOrCreate) \\
&\quad | \ expression\ \texttt{'['}\ expression\ \texttt{']'} \\
&\quad | \ callOrCreate \\
&\quad | \ \texttt{prefix=('+' | '-')}\ expression \\
&\quad | \ expression\ \texttt{bop=('*' | '/' | '\%')}\ expression \\
&\quad | \ expression\ \texttt{bop=('+' | '-')}\ expression \\
&\quad | \ expression\ \texttt{bop=('<=' | '>=' | '<' | '<')}\ expression \\
&\quad | \ expression\ \texttt{bop=('==' | '!=')}\ expression \\
&\quad | \ expression\ \texttt{bop='\&\&'}\ expression \\
&\quad | \ expression\ \texttt{bop='||'}\ expression \\
&\quad | \ \texttt{<assoc=right>}\ expression\ \texttt{bop='?'}\ expression\ \texttt{':'}\ expression
\end{aligned}
$$

```
                | <assoc=right> expression bop=('=' | '+=' | '-=') expression
                | <assoc=right> Identifier bop='->' Identifier
        primary → '(' (listComp)? ')'
                | '[' (listComp)? ']'
                | Identifier
                | literal
       listComp → expression (',' expression)* (',')?
        literal → integerLiteral
                | 'INF'
                | floatLiteral
                | TimeLiteral
                | DateLiteral
 integerLiteral → (DecimalLiteral | HexLiteral | OctLiteral)
   floatLiteral → (FloatLiteral | HexFloatLiteral)
  callOrCreate → Identifier ('(' expressionList? ')')?
expressionList → expression (',' expression)*

     HpartDef → 'Hparticipant' ':' hSuite
       hSuite → NEWLINE INDENT abilityDecl+DEDENT
  abilityDecl → ('nullable')? callOrCreate

      classDef → Identifier classParamClauses classBody
classParamClauses → ('(' classParams ')')*
  classParams → classParam (',' classParam)*
   classParam → ('val' | 'var')? Identifier ':' Identifier ('=' expression)?
    classBody → '{' classBodyDecl+ '}'
classBodyDecl → (filedDecl | methodDecl)
    filedDecl → 'var' Identifier ':' Identifier ('=' expression)?
  methodDecl → 'def' procedure
    procedure → signature (':' Identifier)? '=' procBody
     procBody → (expression | block)
        block → '{' blockStm* '}'
    signature → Identifier ('(' params? ')')*
      params → param (',' param)*
       param → Identifier (':' Identifier)? ('='expression)?
     blockStm → localVarDecl NEWLINE
              | statement NEWLINE
 localVarDecl → 'var' Identifier ('=' expression)?
    statement → block
              | 'if' '(' expression ')' statement ('else' statement)?
              | 'break' NEWLINE
              | 'continue' NEWLINE
              | expression NEWLINE

     funcDecl → 'compute' procedure
```

A.2 Terminator Characteristics

A.2.1 Identifier

An identifier consists of a letter followed by zero or more letters, digits, underscores, and single quotes.

$$Identifier \rightarrow Letter\ LetterOrDigit*$$

fragment $Letter \rightarrow$ `[a-zA-Z_]`
fragment $LetterOrDigit \rightarrow Letter$
| `[0-9]`

A.2.2 TimeLiteral

Time literals represent a point in time using the 24-hour system (A.1), with a minimum precision of 1 minute. There should be no space between the hour and the colon, and the same between the minute and the colon

$$hh : mm \tag{A.1}$$

For example, the following is a legal time literal.

00:12
20:30
23:59

And these things are not legal.

00:62
25:30
24:00

$$TimeLiteral \rightarrow \texttt{[0-2][0-9]':'[0-6][0-9]}$$

A.2.3 DateLiteral

The date literal uses the form yyyy/mm/dd to represent a date, as shown in xx, with the following lexicon.

2023/02/24
1999/03/30

$$DateLiteral \rightarrow [0-9][0-9][0-9][0-9]'/'[0-1][0-9]'/'[0-3][0-9]$$

A.2.4 Coord

Coordinate uses WGS coordinates to represent a physical location, and wgs coordinates are classified as a FloatConst.

$$\begin{aligned} CoordLiteral &\rightarrow \texttt{'('} \ \texttt{','}[+|-]\texttt{FloatConst')'} \\ &\rightarrow (\texttt{'+'} \mid \texttt{'-'})?\ \texttt{FloatConst} \end{aligned}$$

For instance, the **Coord**$(108.911, 34.153)$ represent the location of xi'an, China in WGS.

A.2.5 Numeric Literals

Integer Literal

$$\begin{aligned} DecimalLiteral &\rightarrow (\texttt{'0'} \mid \texttt{[1-9]}(Digits)? \mid \texttt{'\_'}+\ Digits) \\ \text{fragment}\quad Digits &\rightarrow \texttt{[0-9]}\ (\texttt{[0-9\_]*}\ \texttt{[0-9]})? \\ HexLiteral &\rightarrow \texttt{'0'}\ \texttt{[xX]}\ \texttt{[0-9a-fA-F]}\ (\texttt{[0-9a-fA-F\_]*}\ \texttt{[0-9a-fA-F]})? \\ OctLiteral &\rightarrow \texttt{'0'}\ \texttt{'\_'*}\ \texttt{[0-7]}\ (\texttt{[0-7\_]*}\ \texttt{[0-7]})? \end{aligned}$$

Float Literal

$$\begin{aligned} FloatLiteral &\rightarrow (Digits\ \texttt{'.'}\ Digits \mid \texttt{'.'}\ Digits)\ ExponetPart?\ \texttt{[fFdD]?} \\ &\quad \mid Digits\ (ExponetPart\ \texttt{[fFdD]?} \mid \texttt{[fFdD]}) \\ \text{fragment}\quad ExponetPart &\rightarrow \texttt{[eE]}\ \texttt{[+-]?}\ Digits \\ HexFloatLiteral &\rightarrow Prefix\ (HDigits\ \texttt{'.'?} \mid HDigits?\ \texttt{'.'}\ HDigits)\ HexEPart \\ \text{fragment}\quad HDigits &\rightarrow HDigit\ ((HDigit \mid \texttt{'\_'})*\ HDigit)? \\ \text{fragment}\quad HDigit &\rightarrow \texttt{[0-9a-fA-F]} \\ Prefix &\rightarrow \texttt{'0'}\ \texttt{[xX]} \\ HexEPart &\rightarrow \texttt{[pP]}\ \texttt{[+-]?}\ Digits\ \texttt{[fFdD]?} \end{aligned}$$

A.2.6 Indent Related

In CTL, part of the syntax uses indentation to divide blocks of code. The details are handled in the grammar analysis.

Appendix B

Example Applications

This chapter lists some applications that have been developed using CTL.

B.1 Video Doc

The application can extract video summaries.

Listing B.1: Video Doc App

```
CrowdTask VideoDoc {

    Constraints:
        Precedence:
            TakeVideo -> ExtractSummary -> VideoDoc
        Date:
            FromNowTo(2023/3/30)

    ]

    comupte main():Void = {
        CrowdSO.save("./data", ExtractSummary.returnData)
    }

}

CrowdTask TakeVideo {
    Constraints:
        Spatial:
            (108.80, 34.33) - (109.05, 34.20)

    Hparticipant:
        Location
```

```
        LocalTime
        HumanIntelligence
        Video

    comupte main(): Video = Hpart.Video
}

CrowdTask ExtractSummary {
    Hparticipant:
        HumanIntelligence

    comupte main(): = {
        result = CrowdOS.createTask(
            "summary the video",
            TakeVideo.returnData
            )
        result
    }
}
```

B.2 Journalist

The application collects news information in the city.

Listing B.2: Journalist App

```
CrowdTask Journalist {
    Constraints:
        Spatial:
            (108.80, 34.33) - (109.05, 34.20)
        Date:
            FromNowTo(2023/6/30)


    Hparticipant:
        Location
        HumanIntelligence
        image

    compute main(): Void = {
        CrowdOS.save("./data/",(Hpart.Pid, Hpart.Image))
    }
}
```

B.3 Map3D

The application collects 3D coordinate information data in a specified area for building 3D maps.

Listing B.3: Map3D App

```
CrowdTask Map3D {
    Constraints:
        Spatial:
            (108.80, 34.33) - (109.05, 34.20)

    Hparticipant:
        Location

    compute main():Void = {
        CrowdOS.save("./data/Map3D/",(Hpart.Pid, Hpart.Location))
    }
}
```

B.4 Broken Streetlight

The app monitors damaged streetlights in the city and assigns crews to fix them, and records the results.

Listing B.4: Broken Streetlight App

```
CrowdTask BrokenStreetlight {
    Constraints:
        Precedence:
            BrokenStreetlightSense -> BrokenStreetlightAction
            BrokenStreetlightAction -> BrokenStreetlightReview

    CrowdTask BrokenStreetlightSense {
        Constraints:
            Spatial:
                (108.80, 34.33) - (109.05, 34.20)

        Hparticipant:
            Location
            image

        compute main(): BrokenStreetlightInfo = {
            BrokenStreetlightInfo(Hpart.Location, Hpart.Image)
        }
    }

}

```

```
CrowdTask BrokenStreetlightAction {
    Constraints:
        Spatial:
            BrokenStreetlightSense.location
        Priority: 9

    Hparticipant:
        Specific("skill:PowerLineRepair")

    compute main() = {
        CrowdOS.save("./data/BrokenStreetlightLog/")
    }
}

CrowdTask BrokenStreetlightReview {
    Constraints:
        Spatial:
            BrokenStreetlightSense.Location


    Hparticipant:
        Location
        Image

    compute main(): Void = {
        CrowdOS.save(
            "./data/BrokenStreetlightReview/",
            (Hpart.Pid, Hpart.Location, Hpart.Image)
            )
    }
}

class BrokenStreetlightInfo(
    var location: Location,
    var img: Image
){
    def getLocation() = location
    def getImg() = img
}
```

B.5 Missing Manhole

The app monitors lost manhole covers in the city and assigns people to fix them, and records the results.

Listing B.5: Missing Manhole App

```
CrowdTask MissingManholes {
    Constraints:
```

```
        Precedence:
            MissingManholesSense -> MissingManholesAction
            MissingManholesAction -> MissingManholesReview

}

CrowdTask MissingManholesSense {
    Constraints:
        Spatial:
            (108.80, 34.33) - (109.05, 34.20)
    Hparticipant:
        Location
        image

    compute main() = (Hpart.Location, Hpart.Image)
}

CrowdTask MissingManholesAction {
    Constraints:
        Spatial:
            MissingManholesSense.returnData.Location
        Priority: 9

    Hparticipant:
        Specific("skill:RoadRepair")

    compute main(): Void = {
        CrowdOS.save("./data/MissingManholesLog/", Hpart.Pid)
    }
}

CrowdTask MissingManholesReview {
    Constraints:
        Spatial:
            MissingManholesSense.Location

    Hparticipant:
        Image

    compute main():Void = {
        save("path", (Hpart.Pid, Hpart.Location, Hpart.image))
    }
}
```

B.6 Rebalance Bicycle

The app assigns workers to rebalance the distribution of shared bicycles with vehicles.

Listing B.6: Rebalance Bicycle App

```
1
2  CrowdTask ReBalenceBicycle {
3      Constraints:
4          Precedence:
5              RebalenceTrigger -> DispatchVehice
6              RebalenceTrigger -> DispatchWorker
7              DispatchVehice -> RebalenceTask
8              DispatchWorker -> RebalenceTask
9
10     compute main() = 0
11 }
12
13 CrowdTask RebalenceTrigger = CrowdOS.systemTask("Trigger")
14
15 CrowdTask DispatchVehice {
16     Constraints:
17         Spatial:
18             RebalenceTrigger.Location
19
20     Hparticipant:
21         Location
22         Specific("type:Vehice")
23 }
24
25 CrowdTask DispatchWorker {
26     Constraints:
27         Spatial:
28             RebalenceTrigger.Location
29
30     Hparticipant:
31         Location
32         Image
33         HumanIntelligence
34
35     compute main(): void =
36         CrowdOS.save(
37             "./data/DispatchWorker/",
38             (Hpart.Pid, Hpart.Image, Hpart.Location)
39         )
40 }
41
42 CrowdTask RebalenceTask {
43
44     Constraints:
45         Spatial:
46             (108.80, 34.33) - (109.05, 34.20)
47
48     Hparticipant:
49         Image
50         Specific(DispatchWorker.return.Pid)
51
```

```
    compute main(args: List[String]): Void = {
        CrowdOS.save(
            "./data/RebalenceTaskLog/",
            (Hpart.Pid, Hpart.Image, Hpart.Location)
        )
    }

}
```

B.7 Busy Canteen

The app monitors the flow of customers in the canteen at noon.

Listing B.7: Busy Canteen App

```
CrowdTask BusyCanteen {
    Constraints:
        Spatial:
            (108.80, 34.33) - (109.05, 34.20)
        Temporal:
            11:00 - 13:00


    Hparticipant:
        Image
        HumanIntelligence
        Text

    compute main(): Void = {
        if (Hpart.Text.getvalue == "Too Busy"){
            CrowdOS.println("Come back later.")
        } else {
            CrowdOS.println("It's time to eat.")
        }
    }
}
```

B.8 Noise Detection

The app monitors noise on campus.

Listing B.8: Noise Detection App

```
CrowdTask NoiseDetection {
    Constraints:
        Temporal:
```

```
4               8:00 - 12:00
5           Spatial:
6               (108.80, 34.33) - (109.05, 34.20)
7
8       Hparticipant:
9           Microphone,
10
11      compute main() =  {
12          CrowdOS.save("./data/NoiseDetection/",
13              (Hpart.pid, Hpart.Microphone)
14              )
15      }
16  }
```