ISPL Tutorial[*]

Natasha Alechina and Brian Logan

1 Introduction

The practicals and coursework for INFOMLSAI uses the model checker MCMAS. The MCMAS manual contains an overview of ISPL syntax, but you may find this short tutorial on ISPL helpful. (The tutorial applies to both MCMAS versions 1.1.0 and 1.3.0.)

2 Overview of ISPL

MCMAS uses `ispl` files for system and property descriptions. The Interpreted Systems Programming Language (ISPL) is based on the notion of *interpreted systems* introduced in [1]. ISPL is designed for representing and reasoning about multi-agent systems, and allows the *local states* of agents and the environment, and the actions available in a given state, to be compactly specified.

ISPL distinguishes between two kinds of agents: "autonomous" agents, and the environment agent. In what follows, we will refer to "autonomous" agents as simply "agents" and the environment as the "environment agent". The environment agent is used to describe boundary conditions and infrastructure shared by the "autonomous" agents, and is described in a similar way to autonomous agents (see below). As not all system descriptions require an environment, the environment agent is optional in an ISPL program.

Each agent (including the environment, if included) is characterised by:

- a set of *local states* (e.g., the states "ready" or "busy")
- a set of *actions* (e.g., "sendmessage")
- a rule or *protocol* describing which actions can be performed by an agent in a given local state

[*]This tutorial is based on a tutorial by Franco Raimondi, and we thank Franco for permission to use material from his tutorial.

- an *evolution function*, describing how the local states of the agents evolve based on their current local state and on other agents' actions

Local states. Local states are defined in terms of *local variables*. For example, consider a printer with two sensors, one sensor for toner (which could be high or low), and one sensor for paper (which could be full or empty). In this case, the printer agent has four possible local states corresponding to all the possible combinations of values of toner and paper. The local states of agents are private, i.e., each agent can read and update only its own local variables, and its protocol and evolution function cannot read (or update) the local variables of other agents. However each agent i can read a subset of the local variables of the environment agent (if there is an environment), which are termed the *local observable variables* for agent i. The local observable variables for agent i can be referenced by agent i's protocol and evolution function, and form part of its *extended local state*. However, their value can only be changed by the environment. In addition, the epistemic accessibility relation of an agent[1] is based on the agent's extended local states. Intuitively, an agent "knows" something in a state of the system if this something is true in all the states of the system in which its extended local states remain the same. In the rest of the tutorial, we refer to extended local states as simply local states. Different agents may read different local observable variables of the environment (if an environment variable is readable by all agents, it is termed *globally observable*).

A subset of the local states of an agent may optionally be specified to be *red states*. Red states are defined by a Boolean formula over the agent's local variables and local observable variables (local variables for the Environment agent). All local states that satisfy the formula are *red*, and all states that do not satisfy the formula are *green*. Red states effectively allow the definition of an atomic proposition characterising some "bad" or "good' property of the agent that can be used in specifying properties of the multi-agent system as a whole.

Actions. Each agent (including the environment) is allowed to perform some actions, for instance sending a message. It is assumed that all actions performed are visible by all the other agents.

Protocols. Protocols describe which actions can be performed in a given local state. As local states are defined in terms of variables, the protocol for an agent is expressed as a function from variable assignments to sets of actions. In ISPL, protocols are not required to be exhaustive: it is sufficient to specify only the variables

[1]Epistemic accessibility will be covered in Week 3.

assignments relevant to the execution of certain actions, and introduce a catch-all assignment by means of the keyword `Other` (see below). Protocols are also not required to be deterministic: it is possible to associate a set of actions to a given variable assignment. In this case the action to be performed is selected non-deterministically from this set.

Evolution functions. The evolution function for an agent describes how variable assignments change as a results of the actions performed by *all* the other agents. For instance, the evolution function for a printer could prescribe that, if the current local state (or a variable comprising the local state) of the printer agent is “ready” and another agent performs the action “send print job”, then the next local state of the printer agent is “busy”. Formally, the evolution function is a function returning a *next* assignment to the local variables of an agent as a function of the *current* set of assignments to local variables, the observable variables of the environment, and the actions performed by the agents. A global evolution function is computed by taking the conjunction of all the agents’ evolution functions.

In addition to the definition of each agent, the definition of a MAS contains an evaluation and a specification of of the possible initial states of the MAS. The *evaluation* defines propositions that specify global system properties in terms of the local states of the agents. The *initial states* are specified by the declaration of a set of initial states expressed as assignments to local variables. If more than one state satisfies the assignments, then the initial state is selected randomly. The system evolves from this set of initial states in accordance to the protocols and the evolutions functions, and this process is used to compute the truth value of formulae (properties) specified by the user.

3 Syntax of an ISPL File

An ISPL file consists of a number of `Agent` sections, one for each agent (and the environment if there is one). The agent sections are followed by four sections relating to the MAS as a whole: `Evaluation`, `InitStates`, `Groups` and `Formulae`. The `Evaluation` section defines the propositions appearing in formulae (properties) in terms of agent and environment variables. The `InitStates` section defines the initial state(s) of the MAS. The `Groups` section defines which agents are members of which group(s), and the `Formulae` section defines the properties to be checked.

A valid ISPL file must contain at least one `Agent` section, and `Evaluation`, `InitStates` and `Formulae` sections. The `Environment` section can be omitted. If there are no formulae that refer to groups, the `Group` section can also be omitted.

3.1 Defining the Agents and Environment

Each agent is defined in an `Agent` section, delimited by `Agent <name>` and `end Agent`. `<name>` specifies the name of the agent, e.g., `Agent Robot1`. The name `Environment` is reserved, and is used to specify the environment, if there is one. Each `Agent` section consists of four (sub)sections: `Vars`, `Actions`, `Protocol` and `Evolution` as in Section 2. For an agent definition to be valid, each of the `Vars`, `Actions`, `Protocol` and `Evolution` sections must be present and non-empty. However, any definitions in the `Environment` section that are not required for a model can be left blank; for example, if the environment does not change, the definition of the `Evolution` for the Environment can be blank. In addition, the definition of the environment agent may contain an optional `Obsvars` section specifying the variables that are local observable for all agents, and agents may contain `Lobsvars` section, specifying which of the `Vars` of the environment agent can be read by the agent. Finally, both agents and the environment agent may contain an optional `RedStates` section.

Defining variables. An agent's variables are declared and defined in the `Vars` section, delimited by `Vars:` and `end Vars`. There are three types of variables: *Boolean*, *enumeration* and *bounded integer*. The types of variables are implicitly specified by their definition. For example, if `x`, `y` and `z` are variables of Boolean, enumeration and bounded integer respectively, they are be defined as follows:

```
x : boolean;
y : {a, b, c};
z : 1 .. 4;
```

The value of `x` can be *true* or *false*, the value of `y` is one of `a`, `b` and `c`, and the value of `z` can be 1, 2, 3, or 4. Comparisons between Boolean variables and between enumeration variables are restricted to equality tests, e.g., `x = true`, `y = a`, `x != false` or `y != b`. The arithmetic operations "=", "!=", "<", "<=", ">", ">=" can be used with bounded integers, e.g., `z < 2`, `z >= z * 2 - 3`.

The local observable variables for an agent are defined in the section `Lobsvars`; for example:

```
Lobsvars = { a, b };
```

where `a` and `b` are defined in the `Vars` section of the environment agent. Variables that are locally observable for all agents can be defined in an `Obsvars` section in the environment agent; for example;

```
Obsvars = { c };
```

Defining actions. The (names of) actions agent can perform are declared in the `Actions` section. (The effects of actions are specified by the evolution function.) For example:

```
Actions = { a1, b2, c3};
```

Defining the protocol. The actions an agent can perform in a given local state are defined in the `Protocol` section, delimited by `Protocol:` and `end Protocol`. Protocols are defined in relational form. Each line consists of a *condition*, which is a Boolean formula over local states, and a list of actions. The line specifies the actions that can be performed in the local states specified by the condition. For example:

```
x = true and Environment.z < 2 : { a1, a3 };
```

`x = true` and `Environment.z < 2` is the condition, and `{a1, a3}` is the list of actions. The conditions appearing in different lines do not need to be mutually exclusive. If this is the case, the agent has nondeterministic behaviour and all behaviours allowed by the protocol specification are considered possible by MCMAS.

For an agent with many local states, it may be infeasible to specify actions for every state. The (optional) keyword `Other:` in a protocol specification represents that in all local states that *do not* satisfy any of the conditions in the protocol definition, (one of) the specified list of actions my be performed. For example:

```
Other : { action-list };
```

`Other` must occur last in a protocol section, i.e., it encodes all states except those specified in any line appearing before it. The `Other` keyword is also useful if the same set of actions is allowed in all local states; in this case, `Other` is the only item in the protocol section.

Defining the evolution function. Evolution functions are defined in the `Evolution` section, delimited by `Evolution:` and `end Evolution`. As with protocols, evolution functions are defined in relational form. Each line consists of a set of assignments to local variables and an *enabling condition*, which is a Boolean formula over local variables, observable variables of the Environment, and actions of all agents. Each variable assignment must be consistent with the type of the variable being assigned, e.g., only a Boolean value or an expression returning a

Boolean value can be assigned to a boolean variable. Multiple assignments can be connected by the keyword `and`. In an enabling condition, an observable variable must have the prefix `Environment`, e.g., `Environment.z`, and when referring to actions, the name of the agent must be specified, e.g., `Agent1.Action = a1`. This can be simplified to `Action = a1` if `Agent1` is the agent whose evolution function is being defined. For example, an evolution function might contain the line:

```
(x = true and z = Environment.z + 1) if (y = b and Agent1.Action = a1);
```

which is read as: "in the next step, the value of `x` is true and the value of `z` is equal to the (current) value of `z` for the Environment + 1, *if* the current value of `y` is `b` and `Agent1` is performing action `a1`".

A line is *enabled* in a state if its enabling condition is satisfied. If more than one line is enabled, which variables are updated depends on the value of the `Semantics` flag. If `Semantics = MultiAssignment` (the default), a single enabled line is chosen nondeterministically, and the corresponding variable assignment(s) applied. If `Semantics = SingleAssignment`, then all enabled lines that update *disjoint* variables are applied (if two or more enabled lines update the same variable, then one is chosen nondeterministically as for `MultiAssignment`). Note that in a state where no line in an agent's evolution function is enabled, the agent's state is not updated (i.e., the agent stays in the same state) no matter which action it executes.

Defining the red states. Red states are defined in the `RedStates` section, delimited by `RedStates:` and `end RedStates`. The red states of an agent are defined by a Boolean formula over its local variables and local observable variables. For example:

```
x = true and (!(Environment.y = a) or z > 3)
```

The `RedStates` section is optional.

3.2 Defining the MAS

The remaining sections of an ISPL file define the MAS and the properties to be checked.

Defining the evaluation function. Evaluation functions are defined in the `Evaluation` section of an ISPL file, delimited by `Evaluation` and `end Evaluation`. An evaluation function consists of a set of atomic propositions. Each atomic proposition is associated with a Boolean formula over the local variables of all agents and the observable variables of the Environment. Each variable involved in the formula has a prefix indicating the agent the variable belongs to. For example, an atomic proposition may be defined as

```
happy if Environment.x = true and TestAgent.z < Environment.z;
```

where `happy` is an atomic proposition and `if` is a keyword. A proposition evaluates to true in all the global states that satisfy the Boolean formula.

Defining the initial states. Initial states are defined in the `InitStates` section of an ISPL file, delimited by `InitStates` and `end InitStates`. Initial states are defined by a Boolean formula over variables, similarly to the definition of atomic propositions in the evaluation function. However, when specifying the initial states, each proposition in the Boolean formula is of the form:

```
XXX.x = xxx
```

where XXX is an agent or the Environment, `x` is a variable of `XXX` and `xxx` is a truth value, an enumeration value or an integer, depending on the type of the variable. (Note that arithmetic expressions are not allowed when specifying the initial value of variables of integer type.) For example:

```
Environment.x = false and Environment.y = a and
TestAgent.x = true and TestAgent.z = 1;
```

Defining groups. Groups are defined in the `Groups` section of an ISPL file, delimited by `Groups` and `end Groups`. Groups are used in formulae involving group modalities, e.g., Epistemic or ATL formulae. A group is defined as a set of one or more agents, including the Environment, e.g.,

```
g1 = { Robot1, Environment };
```

Defining properties. The properties to be verified are defined in the `Formulae` section of an ISPL file, delimited by `Formulae` and `end Formulae`. A formula is defined in terms of atomic propositions defined in the `Evaluation` section. The syntax of formulae is given by:

⟨*formula*⟩ ::= ‘`(`’ ⟨*formula*⟩ ‘`)`’
  | ⟨*formula*⟩ ‘`and`’ ⟨*formula*⟩
  | ⟨*formula*⟩ ‘`or`’ ⟨*formula*⟩
  | ‘`!`’ ⟨*formula*⟩
  | ⟨*formula*⟩ ‘`->`’ ⟨*formula*⟩

```
|  ‘AG’ ⟨formula⟩
|  ‘EG’ ⟨formula⟩
|  ‘AX’ ⟨formula⟩
|  ‘EG’ ⟨formula⟩
|  ‘AF’ ⟨formula⟩
|  ‘EF’ ⟨formula⟩
|  ‘A’ ‘(’ ⟨formula⟩ ‘U’ ⟨formula⟩ ‘)’
|  ‘E’ ‘(’ ⟨formula⟩ ‘U’ ⟨formula⟩ ‘)’
|  ‘K’ ‘(’ AgentName ‘,’ ⟨formula⟩ ‘)’
|  ‘GK’ ‘(’ GroupName ‘,’ ⟨formula⟩ ‘)’
|  ‘GCK’ ‘(’ GroupName ‘,’ ⟨formula⟩ ‘)’
|  ‘DK’ ‘(’ GroupName ‘,’ ⟨formula⟩ ‘)’
|  ‘O’ ‘(’ AgentName ‘,’ ⟨formula⟩ ‘)’
|  ‘<’ GroupName ‘>’ ‘X’ ⟨formula⟩
|  ‘<’ GroupName ‘>’ ‘F’ ⟨formula⟩
|  ‘<’ GroupName ‘>’ ‘G’ ⟨formula⟩
|  ‘<’ GroupName ‘>’ ‘(’ ⟨formula⟩ ‘U’ ⟨formula⟩ ‘)’
|  ‘AtomicProposition’
```

where AgentName is the name of an agent or the Environment, GroupName is the name of a group defined in the `Group` section, and AtomicProposition is AgentName`.RedStates`, AgentName`.GreenStates`, or an atomic proposition defined in the `Evaluation` section.

References

[1] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.