Simple Tutorial for the CSP-Z3 Model Checker

Kun Wei

June 2018

1 Installation and running environment

To implement the CSP-Z3 model checker, users need to install the latest version of Z3Py (and Python 3.x), which can be downloaded from

https://github.com/Z3Prover/z3/wiki/Using-Z3Py-on-Windows

The users just need to unzip the files to anywhere of your computer as long as the Z3py can be found by Python whenever the model checker is executed. Although we developed this model checker with Windows, but it should run with other operating systems as well, because we provide Z3-supported Python files.

There are four files only in this model checker. The `init.py` is a configuration file in which often defines the processes identifications. The `finite_set.py` is a set theory and `list.py` is a list theory. Finally, `csp.py` is the script for CSP operators and properties.

2 The standard CSP

We have not provide a tool to translate any CSP or *Circus* scripts into our Z3 scripts. So, we have to work on the code level at the moment. The users need to produce two files; one is the file for defining events including the finite sets, the other is to construct CSP processes, refinement or other properties.

2.1 Events

The users can give their own file names or the default one (event.py). The event file must be included in the list.py. We use abstract DataType to define events, and of course the users can use any other DataTypes. A collection of simple events can be defined as

```
Event, (a,b,c) = EnumSort('Event', ('a','b','c'))
```

For compound events such as in.1 or out.1, the users need to define in and out first and then use them to define a tuple. For example,

```
Channel, (in, out) = EnumSort('Channel', ('in', 'out'))
Event = Datatype('Event')
Event.declare('CE', ('channel', Channel), ('value', IntSort()))
Event = Event.create()
```

Often the users need to define a `SetSort` for declaring a set variable, and a `Set` for constructing a finite set for describing refusal sets.

```
SetSort = FSetSort([a,b,c])
Set = FSetDecl([a,b,c])
```

It is also convenient to define a syntax sugar for `Fullset` as

```
Fullset = Set.fullset()
```

Finally, this file must be included in the list.py file so that various functions of the list theory can use the defined events. Note that the names like *Event*, *SetSort*, *Set* and *Fullset* are reserved by the model checker.

2.2 CSP processes

The users can directly use `Chaos`, `Miracle`, `Skip` and `Stop` in the scripts for representing the corresponded CSP primitives.

We use a simple prefix and sequential composition to define prefix. For example, the process $P = a \to b \to Skip$ is expressed as

```
P = Seq(SP(a), SP(b))
```

Similarly, $P = a \to b \to Stop$ is expressed as `P = Seq(Seq(SP(a),SP(b)),Stop)` or `P = Seq(SP(a), Seq(SP(b), Stop))`.

In Z3 we use `EC(P,Q)` and `IC(P,Q)` to represent external and internal choices. For example, `EC(SP(a), SP(b))` represents $a \to Skip \;\square\; b \to Skip$. Parallel composition, e.g., $a \to b \to Skip \underset{\{b\}}{\parallel} b \to c \to Skip$ is expressed as

```
P = Par( [ b ], Seq(SP(a), SP(b)), Seq(SP(b), SP(c)) )
```

Hiding, e.g., $(a \to b \to Skip) \setminus a$ is expressed as

```
P = Hide( Seq(SP(a), SP(b)), [ a ])
```

Recursive processes are constructed differently from non-recursive processes. The users first define the body of the recursive process including the variables for invocation and create the objects, and then set up the objects with the recursive variables, and finally pass the objects to specific processes. For example, mutually recursive processes, $P = a \to P \;\square\; b \to Q$ and $Q = a \to P \;\square\; b \to Q$, are defined as

```
X = RecP( 'EC( Seq(SP(a), X), Seq(SP(b), Y))', 1)
Y = RecP( 'EC( Seq(SP(a), X), Seq(SP(b), Y))', 1)
X.setup([ 'X', 'Y' ], [X, Y])
Y.setup(['X', 'Y'], [X, Y])
P = X.create()
Q = Y.create()
```

2.3 Refinement

There are many auxiliary functions we have defined, which can show some observations of a CSP process. For example, `ListOneTerminatedTrace()`, `ListAllTerminatedTraces()`, `ListAllTraces()` and `ListAllTracesAndRefs()` can be understood just by their names. Similar to CSP/FDR, we have three models as well in case we dont always consider refinement at the most complex situation. Therefore, `TRef(P,Q)`, `SFRef(P,Q)` and `FDRef(P,Q)` represent the refinement of P and Q in the traces model, the stable failures model and the failures and divergences model. There are other properties such as `DLF(P)` and `DVF(P)` to represent the checking for deadlock freedom and divergence freedom respectively.

3 CSP with local variables

TBC

4 CSP with shared variables

TBC