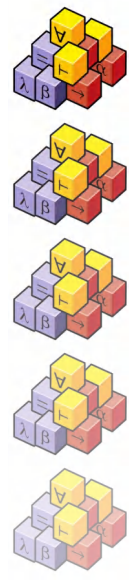
Z Machines

Simon Foster Jim Woodcock
University of York

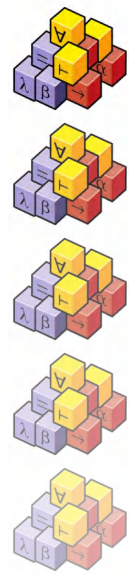
20th August 2022

Overview

1. Modelling Z Specifications with Z Machines
2. Types, Stores, and State Variables
3. Operations and Machines
4. Animation and Verification
5. Managing Requirements



Overview



Motivation

- The Z notation is a rich specification language:

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

- The Z notation is a rich specification language:

IncubatorMonitor

$temp : \mathbb{Z}$

$MIN \leq temp \leq MAX$

Increment

$\Delta IncubatorMonitor$

$temp < MAX$

$temp' = temp + 1$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

- The Z notation is a rich specification language:

$$
\begin{array}{|l}
\hline
IncubatorMonitor \\
temp : \mathbb{Z} \\
\hline
MIN \leq temp \leq MAX \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
Increment \\
\Delta IncubatorMonitor \\
\hline
temp < MAX \\
temp' = temp + 1 \\
\hline
\end{array}
$$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

- The Z notation is a rich specification language:

$$\begin{array}{|l} IncubatorMonitor \\ temp : \mathbb{Z} \\ \hline MIN \leq temp \leq MAX \end{array}$$

$$\begin{array}{|l} Increment \\ \Delta IncubatorMonitor \\ \hline temp < MAX \\ temp' = temp + 1 \end{array}$$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

- The Z notation is a rich specification language:

$$\begin{array}{|l} IncubatorMonitor \\ temp : \mathbb{Z} \\ \hline MIN \leq temp \leq MAX \end{array}$$

$$\begin{array}{|l} Increment \\ \Delta IncubatorMonitor \\ \hline temp < MAX \\ temp' = temp + 1 \end{array}$$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

- The Z notation is a rich specification language:

$$\begin{array}{|l} IncubatorMonitor \\ temp : \mathbb{Z} \\ \hline MIN \leq temp \leq MAX \end{array}$$

$$\begin{array}{|l} Increment \\ \Delta IncubatorMonitor \\ \hline temp < MAX \\ temp' = temp + 1 \end{array}$$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

- The Z notation is a rich specification language:

$$\begin{array}{|l} IncubatorMonitor \\ temp : \mathbb{Z} \\ \hline MIN \leq temp \leq MAX \end{array}$$

$$\begin{array}{|l} Increment \\ \Delta IncubatorMonitor \\ \hline temp < MAX \\ temp' = temp + 1 \end{array}$$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

- The Z notation is a rich specification language:

$$\begin{array}{|l} IncubatorMonitor \\ temp : \mathbb{Z} \\ \hline MIN \leq temp \leq MAX \end{array}$$

$$\begin{array}{|l} Increment \\ \Delta IncubatorMonitor \\ \hline temp < MAX \\ temp' = temp + 1 \end{array}$$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Motivation

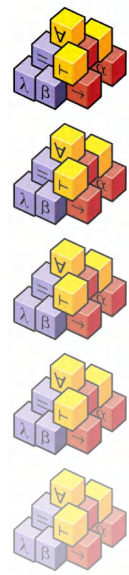
- The Z notation is a rich specification language:

$$\begin{array}{|l} IncubatorMonitor \\ temp : \mathbb{Z} \\ \hline MIN \leq temp \leq MAX \end{array}$$

$$\begin{array}{|l} Increment \\ \Delta IncubatorMonitor \\ \hline temp < MAX \\ temp' = temp + 1 \end{array}$$

- This flexibility can however make it difficult to automate in practice.
- Z Machines: restricted subset of Z in Isabelle/HOL (like Event-B).
- A design pattern for the more general Z language.
- Closer to an implementation and requires some design decisions.
- Careful handling of types vs. set dichotomy, which is less visible in Z.
- Includes support for (1) proof obligation generation and (2) animation.

Overview



Example: Dwarf Signal Types (Z)

$$LampId ::= L1 \mid L2 \mid L3$$

$$dark, stop, warning, drive : \mathbb{F}\, LampId$$

$$dark = \emptyset$$
$$stop = \{L1, L2\}$$
$$warning = \{L1, L3\}$$
$$drive = \{L2, L3\}$$

$$ProperState == \{dark, stop, warning, drive\}$$

$$Signal == \mathbb{F}\, LampId$$

Example: Dwarf Signal Types (Z)

$$LampId ::= L1 \mid L2 \mid L3$$

$$dark, stop, warning, drive : \mathbb{F}\, LampId$$

$$dark = \emptyset$$
$$stop = \{L1, L2\}$$
$$warning = \{L1, L3\}$$
$$drive = \{L2, L3\}$$

$$ProperState == \{dark, stop, warning, drive\}$$

$$Signal == \mathbb{F}\, LampId$$

Example: Dwarf Signal Types (Z)

$$LampId ::= L1 \mid L2 \mid L3$$

$$dark, stop, warning, drive : \mathbb{F}\, LampId$$

$$dark = \emptyset$$
$$stop = \{L1, L2\}$$
$$warning = \{L1, L3\}$$
$$drive = \{L2, L3\}$$

$$ProperState == \{dark, stop, warning, drive\}$$

$$Signal == \mathbb{F}\, LampId$$

Example: Dwarf Signal Types (Z)

$$LampId ::= L1 \mid L2 \mid L3$$

$$dark, stop, warning, drive : \mathbb{F}\, LampId$$

$$dark = \emptyset$$
$$stop = \{L1, L2\}$$
$$warning = \{L1, L3\}$$
$$drive = \{L2, L3\}$$

$$ProperState == \{dark, stop, warning, drive\}$$

$$Signal == \mathbb{F}\, LampId$$

Example: Dwarf Signal Types (Z)

$$LampId ::= L1 \mid L2 \mid L3$$

$$dark, stop, warning, drive : \mathbb{F}\, LampId$$

$$dark = \emptyset$$
$$stop = \{L1, L2\}$$
$$warning = \{L1, L3\}$$
$$drive = \{L2, L3\}$$

$$ProperState == \{dark, stop, warning, drive\}$$

$$Signal == \mathbb{F}\, LampId$$

Example: Dwarf Signal Types (Isabelle)

Example: Dwarf Signal Types (Isabelle)

```
enumtype LampId = L1 | L2 | L3

type_synonym Signal = "LampId set"

enumtype ProperState = dark | stop | warning | drive

definition "ProperState = {dark, stop, warning, drive}"

fun signalLamps :: "ProperState ⇒ LampId set" where
  "signalLamps dark = {}" |
  "signalLamps stop = {L1, L2}" |
  "signalLamps warning = {L1, L3}" |
  "signalLamps drive = {L2, L3}"
```

Example: Dwarf Signal Types (Isabelle)

```
enumtype LampId = L1 | L2 | L3

type_synonym Signal = "LampId set"

enumtype ProperState = dark | stop | warning | drive

definition "ProperState = {dark, stop, warning, drive}"

fun signalLamps :: "ProperState ⇒ LampId set" where
  "signalLamps dark = {}" |
  "signalLamps stop = {L1, L2}" |
  "signalLamps warning = {L1, L3}" |
  "signalLamps drive = {L2, L3}"
```

Example: Dwarf Signal Types (Isabelle)

```
enumtype LampId = L1 | L2 | L3

type_synonym Signal = "LampId set"

enumtype ProperState = dark | stop | warning | drive

definition "ProperState = {dark, stop, warning, drive}"

fun signalLamps :: "ProperState ⇒ LampId set" where
  "signalLamps dark = {}" |
  "signalLamps stop = {L1, L2}" |
  "signalLamps warning = {L1, L3}" |
  "signalLamps drive = {L2, L3}"
```

Example: Dwarf Signal Types (Isabelle)

```
enumtype LampId = L1 | L2 | L3

type_synonym Signal = "LampId set"

enumtype ProperState = dark | stop | warning | drive

definition "ProperState = {dark, stop, warning, drive}"

fun signalLamps :: "ProperState ⇒ LampId set" where
  "signalLamps dark = {}" |
  "signalLamps stop = {L1, L2}" |
  "signalLamps warning = {L1, L3}" |
  "signalLamps drive = {L2, L3}"
```

Example: Dwarf Signal Types (Isabelle)

```
enumtype LampId = L1 | L2 | L3

type_synonym Signal = "LampId set"

enumtype ProperState = dark | stop | warning | drive

definition "ProperState = {dark, stop, warning, drive}"

fun signalLamps :: "ProperState ⇒ LampId set" where
  "signalLamps dark = {}" |
  "signalLamps stop = {L1, L2}" |
  "signalLamps warning = {L1, L3}" |
  "signalLamps drive = {L2, L3}"
```

Stores

- A store is a set of variable declarations to be present in the state.
- Corresponds to a state schema in Z (no dashed variables).
- Introduces several state components (`x`, `y`, `z`) and invariants.
- Invariants can be optionally named and are collected in `state_inv`.

Stores

- A store is a set of variable declarations to be present in the state.
- Corresponds to a state schema in Z (no dashed variables).
- Introduces several state components (`x`, `y`, `z`) and invariants.
- Invariants can be optionally named and are collected in `state_inv`.

Stores

- A store is a set of variable declarations to be present in the state.

```
zstore state =
  x :: T1
  y :: T2
  z :: T3
  where inv1: "assert"
```

- Corresponds to a state schema in Z (no dashed variables).
- Introduces several state components (`x`, `y`, `z`) and invariants.
- Invariants can be optionally named and are collected in `state_inv`.

Stores

- A store is a set of variable declarations to be present in the state.

```
zstore state =
  x :: T1
  y :: T2
  z :: T3
  where inv1: "assert"
```

- Corresponds to a state schema in Z (no dashed variables).
- Introduces several state components (`x`, `y`, `z`) and invariants.
- Invariants can be optionally named and are collected in `state_inv`.

Stores

- A store is a set of variable declarations to be present in the state.

```
zstore state =
  x :: T1
  y :: T2
  z :: T3
  where inv1: "assert"
```

- Corresponds to a state schema in Z (no dashed variables).
- Introduces several state components (`x`, `y`, `z`) and invariants.
- Invariants can be optionally named and are collected in `state_inv`.

Stores

- A store is a set of variable declarations to be present in the state.

```
zstore state =
  x :: T1
  y :: T2
  z :: T3
  where inv1: "assert"
```

- Corresponds to a state schema in Z (no dashed variables).
- Introduces several state components (`x`, `y`, `z`) and invariants.
- Invariants can be optionally named and are collected in `state_inv`.

State Schema and ZStore

State Schema and ZStore

Dwarf

$$last\_proper\_state : ProperState$$
$$turn\_off, turn\_on : \mathbb{F}\ LampId$$
$$last\_state, current\_state : Signal$$
$$desired\_proper\_state : ProperState$$

$$(current\_state \setminus turn\_off) \cup turn\_on = desired\_proper\_state$$
$$turn\_off \cap turn\_on = \emptyset$$

State Schema and ZStore

Dwarf

$$
\begin{array}{l}
last\_proper\_state : ProperState \\
turn\_off, turn\_on : \mathbb{F}\ LampId \\
last\_state, current\_state : Signal \\
desired\_proper\_state : ProperState \\
\hline
(current\_state \setminus turn\_off) \cup turn\_on = desired\_proper\_state \\
turn\_off \cap turn\_on = \emptyset
\end{array}
$$

```
zstore Dwarf =
  last_proper_state :: "ProperState"
  turn_off :: "LampId set"
  turn_on  :: "LampId set"
  last_state :: "Signal"
  current_state :: "Signal"
  desired_proper_state :: "ProperState"
where
  "(current_state - turn_off) ∪ turn_on = signalLamps desired_proper_state"
  "turn_on ∩ turn_off = {}"
```

Dwarf Signal Requirements Schema

Dwarf Signal Requirements Schema

NeverShowAll

Dwarf

$$current\_state \neq \{L1, L2, L3\}$$

Dwarf Signal Requirements Schema

NeverShowAll

Dwarf

$current\_state \neq \{L1, L2, L3\}$

DwarfSignal

NeverShowAll

MaxOneLampChange

ForbidStopToDrive

DarkOnlyToStop

DarkOnlyFromStop

Dwarf Signal Requirements Schema

NeverShowAll

Dwarf

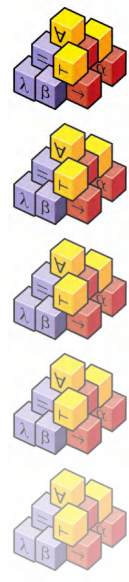
$$current\_state \neq \{L1, L2, L3\}$$

DwarfSignal

NeverShowAll
MaxOneLampChange
ForbidStopToDrive
DarkOnlyToStop
DarkOnlyFromStop

We deal with these separately in Z Machines.

Overview



Operations

- zoperation corresponds to a Z operation schema, but restricted.
- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.
- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.

```
zoperation Op1 =
  over state
  params p1∈"set1" ... pm∈"setm"
  pre "assert"
  update "[x′ = exp, ..., z′ = exp]"
```

- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.

```
zoperation Op1 =
  over state
  params p1∈"set1" ... pm∈"setm"
  pre "assert"
  update "[x′ = exp, ..., z′ = exp]"
```

- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.

```
zoperation Op1 =
  over state
  params p1∈"set1" ... pm∈"setm"
  pre "assert"
  update "[x′ = exp, ..., z′ = exp]"
```

- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.

```
zoperation Op1 =
  over state
  params p1∈"set1" ... pm∈"setm"
  pre "assert"
  update "[x′ = exp, ..., z′ = exp]"
```

- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.

```
zoperation Op1 =
  over state
  params p1∈"set1" ... pm∈"setm"
  pre "assert"
  update "[x′ = exp, ..., z′ = exp]"
```

- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.

```
zoperation Op1 =
  over state
  params p1∈"set1" ... pm∈"setm"
  pre "assert"
  update "[x′ = exp, ..., z′ = exp]"
```

- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operations

- zoperation corresponds to a Z operation schema, but restricted.

```
zoperation Op1 =
  over state
  params p1∈"set1" ... pm∈"setm"
  pre "assert"
  update "[x′ = exp, ..., z′ = exp]"
```

- Inputs ($a?$) and outputs ($b!$) are modelled by parameters.
- Parameters come from sets, not types. Can depend on the state.
- Preconditions can depend only on the undashed before state.
- Assignments update variables. Unmentioned variables unchanged.
- State invariants are not imposed; we need to prove they are preserved.
- Much closer to an implementation of the model.

Operation to Set New Proper State

Operation to Set New Proper State

SetNewProperState

$\Delta DwarfSignal$

$st? : ProperState$

$current\_state = desired\_proper\_state$

$st? \neq current\_state$

$last\_proper\_state' = current\_state$

$turn\_off' = current\_state \setminus st?$

$turn\_on' = st? \setminus current\_state$

$last\_state' = current\_state$

$current\_state' = current\_state$

$desired\_proper\_state' = st?$

Operation to Set New Proper State

SetNewProperState

$\Delta DwarfSignal$

$st? : ProperState$

$current\_state = desired\_proper\_state$

$st? \neq current\_state$

$last\_proper\_state' = current\_state$

$turn\_off' = current\_state \setminus st?$

$turn\_on' = st? \setminus current\_state$

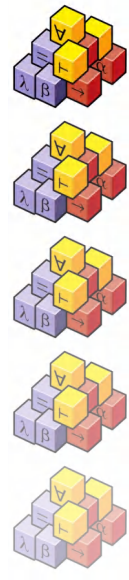
$last\_state' = current\_state$

$current\_state' = current\_state$

$desired\_proper\_state' = st?$

```
zoperation SetNewProperState =
  over Dwarf
  params st∈"ProperState -
              {desired_proper_state}"
  pre "current_state =
         signalLamps desired_proper_state"
  update "[
    last_proper_state' =
      desired_proper_state
    ,turn_off' =
      current_state - signalLamps st
    ,turn_on' =
      signalLamps st - current_state
    ,last_state' =
      current_state
    ,desired_proper_state' = st]"
```

Overview



Machines and Animation

- Often implicit in the Z notation; the set of all operations.
- Generates an interaction tree semantics, for animation.
- Animate using `animate` command. Useful for design-space exploration.
- Sets initial state, checks which operations + parameters are enabled.
- For animation, parameters should typically be drawn from a finite set.

Machines and Animation

- Often implicit in the Z notation; the set of all operations.
- Generates an interaction tree semantics, for animation.
- Animate using `animate` command. Useful for design-space exploration.
- Sets initial state, checks which operations + parameters are enabled.
- For animation, parameters should typically be drawn from a finite set.

Machines and Animation

- Often implicit in the Z notation; the set of all operations.

```
zmachine machine =
  over state
  init "[x′ = exp, ..., z′ = exp]"
  operations Op1 Op2 ... Opn

animate machine
```

- Generates an interaction tree semantics, for animation.
- Animate using `animate` command. Useful for design-space exploration.
- Sets initial state, checks which operations + parameters are enabled.
- For animation, parameters should typically be drawn from a finite set.

Machines and Animation

- Often implicit in the Z notation; the set of all operations.

```
zmachine machine =
  over state
  init "[x′ = exp, ..., z′ = exp]"
  operations Op1 Op2 ... Opn

animate machine
```

- Generates an interaction tree semantics, for animation.
- Animate using `animate` command. Useful for design-space exploration.
- Sets initial state, checks which operations + parameters are enabled.
- For animation, parameters should typically be drawn from a finite set.

Machines and Animation

- Often implicit in the Z notation; the set of all operations.

```
zmachine machine =
  over state
  init "[x′ = exp, ..., z′ = exp]"
  operations Op1 Op2 ... Opn

animate machine
```

- Generates an interaction tree semantics, for animation.
- Animate using `animate` command. Useful for design-space exploration.
- Sets initial state, checks which operations + parameters are enabled.
- For animation, parameters should typically be drawn from a finite set.

Machines and Animation

- Often implicit in the Z notation; the set of all operations.

```
zmachine machine =
  over state
  init "[x′ = exp, ..., z′ = exp]"
  operations Op1 Op2 ... Opn

animate machine
```

- Generates an interaction tree semantics, for animation.
- Animate using `animate` command. Useful for design-space exploration.
- Sets initial state, checks which operations + parameters are enabled.
- For animation, parameters should typically be drawn from a finite set.

Machines and Animation

- Often implicit in the Z notation; the set of all operations.

```
zmachine machine =
  over state
  init "[x′ = exp, ..., z′ = exp]"
  operations Op1 Op2 ... Opn

animate machine
```

- Generates an interaction tree semantics, for animation.
- Animate using `animate` command. Useful for design-space exploration.
- Sets initial state, checks which operations + parameters are enabled.
- For animation, parameters should typically be drawn from a finite set.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. $DwarfSignal$).
- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. $DwarfSignal$).
- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Verification and Proof Obligations

- Requirements specified as machine invariants (cf. *DwarfSignal*).

```
lemma Init_inv: "Init establishes state_inv"

lemma Op1_inv: "Op1 (x, y, z) preserves state_inv"
```

- Machine verification requires we demonstrate:
  1. The initial state assignment establishes the invariants.
  2. Each operation preserves the invariants.
- Specified as Hoare conjectures: $\{state\_inv\}\ Op1(x, y, z)\ \{state\_inv\}$.
- Proof obligations can be generated using method `zpog_full`.
- Weakest preconditions; every operation is a constrained assignment.
- For large models, manage proof obligations using Isar and `explore`.

Example: Dwarf Operations

```
lemma "Init establishes Dwarf_inv"
  by zpog_full

lemma "(SetNewProperState p) preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOn l preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOff l preserves Dwarf_inv"
  by (zpog_full; auto)
```

Example: Dwarf Operations

```
lemma "Init establishes Dwarf_inv"
  by zpog_full

lemma "(SetNewProperState p) preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOn l preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOff l preserves Dwarf_inv"
  by (zpog_full; auto)
```

Example: Dwarf Operations

```
lemma "Init establishes Dwarf_inv"
  by zpog_full

lemma "(SetNewProperState p) preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOn l preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOff l preserves Dwarf_inv"
  by (zpog_full; auto)
```

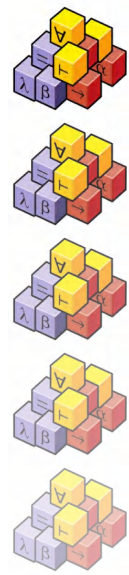
Example: Dwarf Operations

```
lemma "Init establishes Dwarf_inv"
  by zpog_full

lemma "(SetNewProperState p) preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOn l preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOff l preserves Dwarf_inv"
  by (zpog_full; auto)
```

Example: Dwarf Operations

```
lemma "Init establishes Dwarf_inv"
  by zpog_full

lemma "(SetNewProperState p) preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOn l preserves Dwarf_inv"
  by (zpog_full; auto)

lemma "TurnOff l preserves Dwarf_inv"
  by (zpog_full; auto)
```

Overview



Managing Requirements

- Requirements characterised by named assertions using `zexpr`.
- System documentation and safety argumentation.
- Show that each operation preserves safety invariants (or fails to).
- `Req1Failed` is a litmus test for `req1`; enabled if `req1` fails.
- Test using animator. Verify using proof obligation generator.

Managing Requirements

- Requirements characterised by named assertions using `zexpr`.
- System documentation and safety argumentation.
- Show that each operation preserves safety invariants (or fails to).
- `Req1Failed` is a litmus test for `req1`; enabled if `req1` fails.
- Test using animator. Verify using proof obligation generator.

Managing Requirements

- Requirements characterised by named assertions using `zexpr`.

```
zexpr req1 is "assert"
zexpr req2 is "assert"

zoperation Req1Failed =
  over state
  pre "¬ req1"
```

- System documentation and safety argumentation.
- Show that each operation preserves safety invariants (or fails to).
- `Req1Failed` is a litmus test for `req1`; enabled if `req1` fails.
- Test using animator. Verify using proof obligation generator.

Managing Requirements

- Requirements characterised by named assertions using `zexpr`.

```
zexpr req1 is "assert"
zexpr req2 is "assert"

zoperation Req1Failed =
  over state
  pre "¬ req1"
```

- System documentation and safety argumentation.
- Show that each operation preserves safety invariants (or fails to).
- `Req1Failed` is a litmus test for `req1`; enabled if `req1` fails.
- Test using animator. Verify using proof obligation generator.

Managing Requirements

- Requirements characterised by named assertions using `zexpr`.

```
zexpr req1 is "assert"
zexpr req2 is "assert"

zoperation Req1Failed =
  over state
  pre "¬ req1"
```

- System documentation and safety argumentation.
- Show that each operation preserves safety invariants (or fails to).
- `Req1Failed` is a litmus test for `req1`; enabled if `req1` fails.
- Test using animator. Verify using proof obligation generator.

Managing Requirements

- Requirements characterised by named assertions using `zexpr`.

```
zexpr req1 is "assert"
zexpr req2 is "assert"

zoperation Req1Failed =
  over state
  pre "¬ req1"
```

- System documentation and safety argumentation.
- Show that each operation preserves safety invariants (or fails to).
- `Req1Failed` is a litmus test for `req1`; enabled if `req1` fails.
- Test using animator. Verify using proof obligation generator.

Managing Requirements

- Requirements characterised by named assertions using `zexpr`.

```
zexpr req1 is "assert"
zexpr req2 is "assert"

zoperation Req1Failed =
  over state
  pre "¬ req1"
```

- System documentation and safety argumentation.
- Show that each operation preserves safety invariants (or fails to).
- `Req1Failed` is a litmus test for `req1`; enabled if `req1` fails.
- Test using animator. Verify using proof obligation generator.

Example: Dwarf Signal Requirements

Example: Dwarf Signal Requirements

```
zexpr NeverShowAll
  is "current_state ≠ {L1, L2, L3}"
```

Example: Dwarf Signal Requirements

```
zexpr NeverShowAll
  is "current_state ≠ {L1, L2, L3}"

zexpr ForbidStopToDrive
  is "(last_proper_state = stop ⟶ desired_proper_state
     ≠ drive)"
```

Example: Dwarf Signal Requirements

```
zexpr NeverShowAll
  is "current_state ≠ {L1, L2, L3}"

zexpr ForbidStopToDrive
  is "(last_proper_state = stop ⟶ desired_proper_state
     ≠ drive)"

lemma "(SetNewProperState p) preserves NeverShowAll"
  by zpog_full
```

Example: Dwarf Signal Requirements

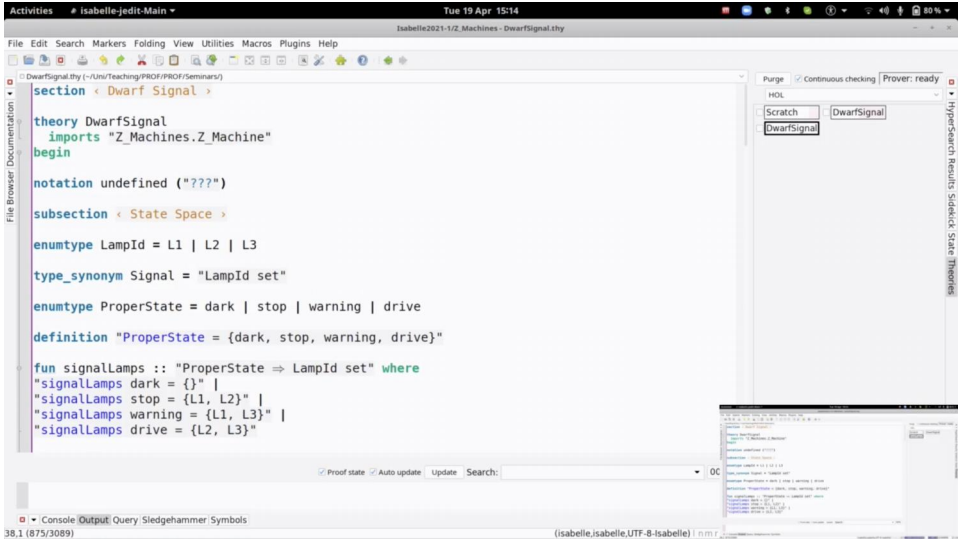
```
zexpr NeverShowAll
  is "current_state ≠ {L1, L2, L3}"

zexpr ForbidStopToDrive
  is "(last_proper_state = stop ⟶ desired_proper_state
     ≠ drive)"

lemma "(SetNewProperState p) preserves NeverShowAll"
  by zpog_full

lemma "TurnOn l preserves NeverShowAll"
  apply zpog_full
  quickcheck
```

Example: Dwarf Signal Requirements

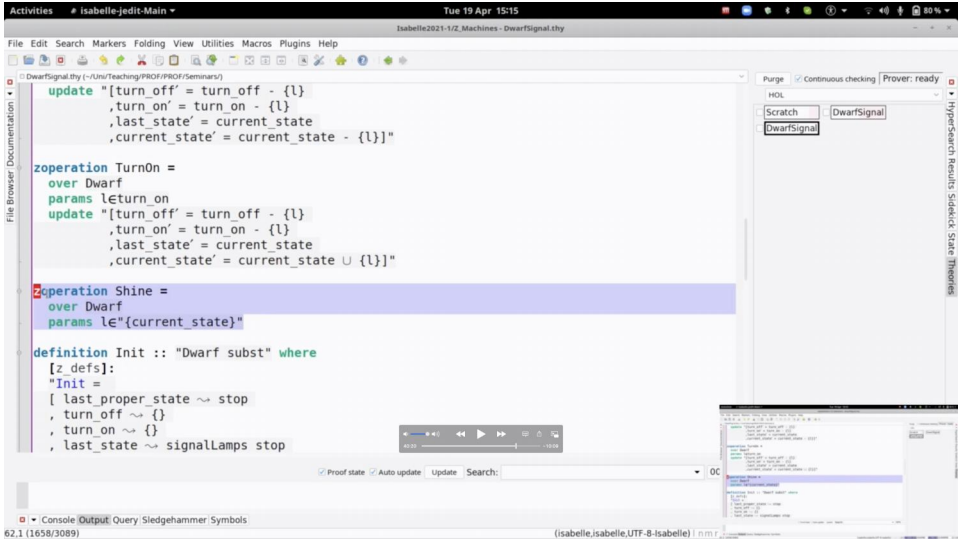
```
zexpr NeverShowAll
  is "current_state ≠ {L1, L2, L3}"

zexpr ForbidStopToDrive
  is "(last_proper_state = stop ⟶ desired_proper_state
     ≠ drive)"

lemma "(SetNewProperState p) preserves NeverShowAll"
  by zpog_full

lemma "TurnOn l preserves NeverShowAll"
  apply zpog_full
  quickcheck

lemma "(SetNewProperState p) preserves ForbidStopToDrive"
  apply zpog_full
  quickcheck
```

Checking the Specification

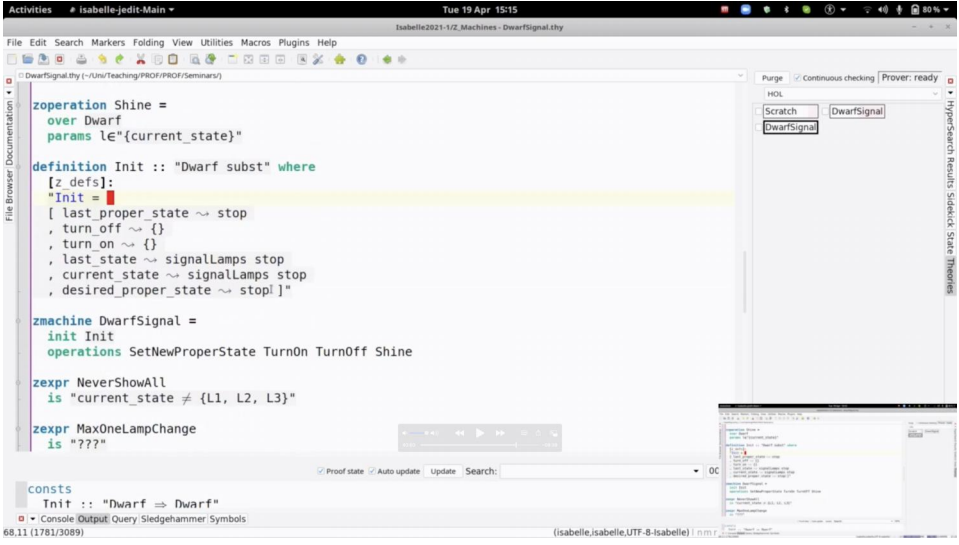
Checking the Specification



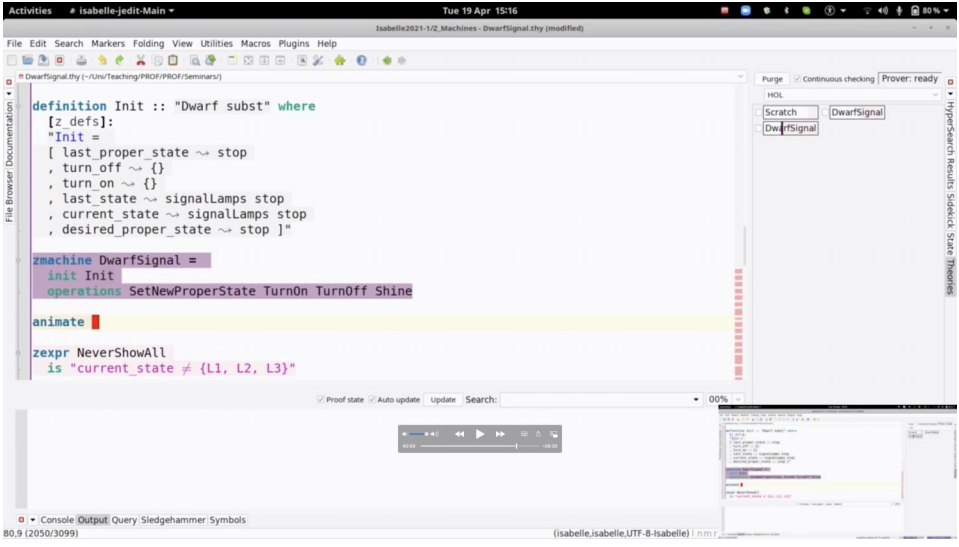
Checking the Specification



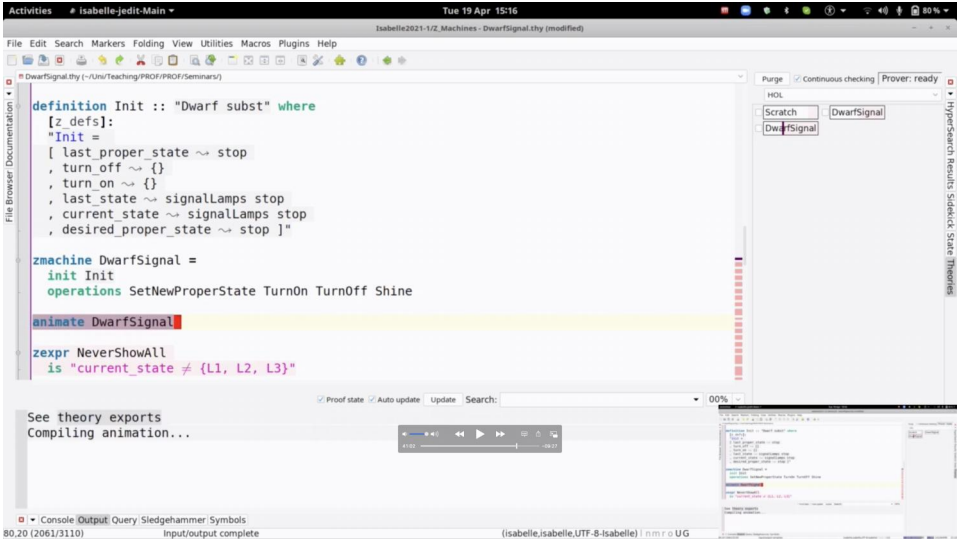
Checking the Specification



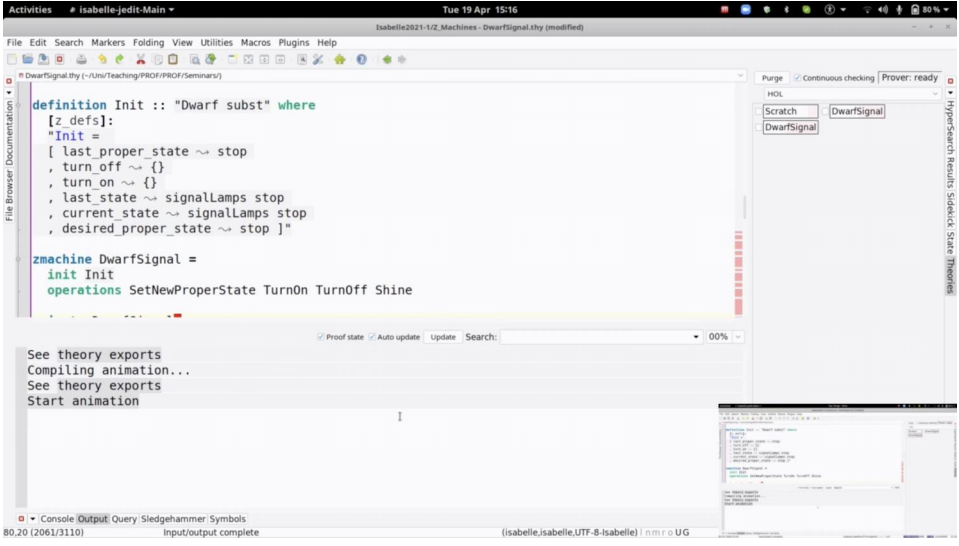
Checking the Specification



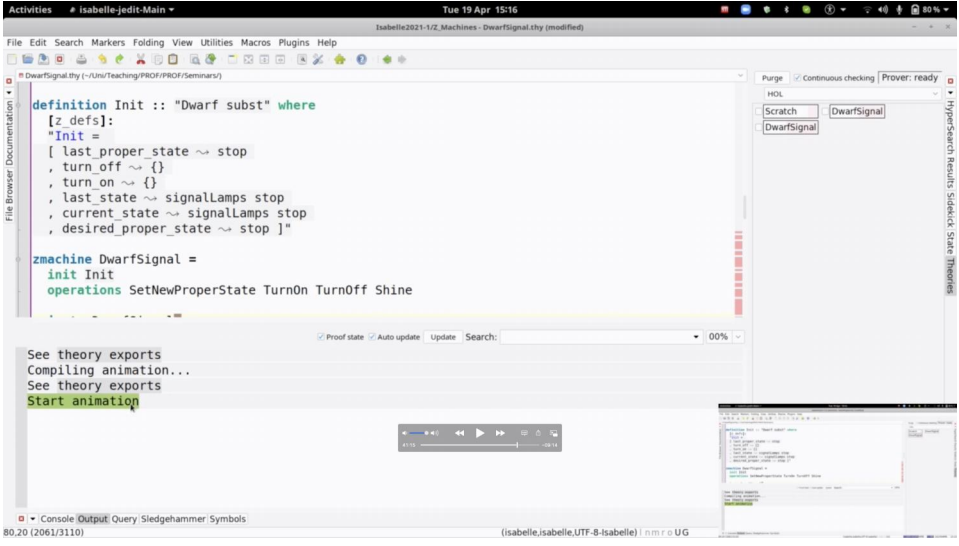
Checking the Specification



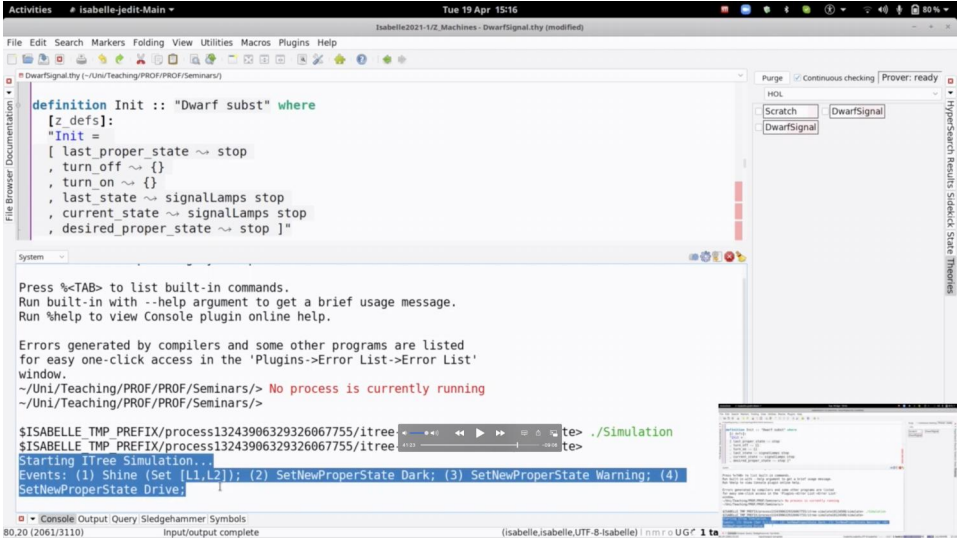
Checking the Specification



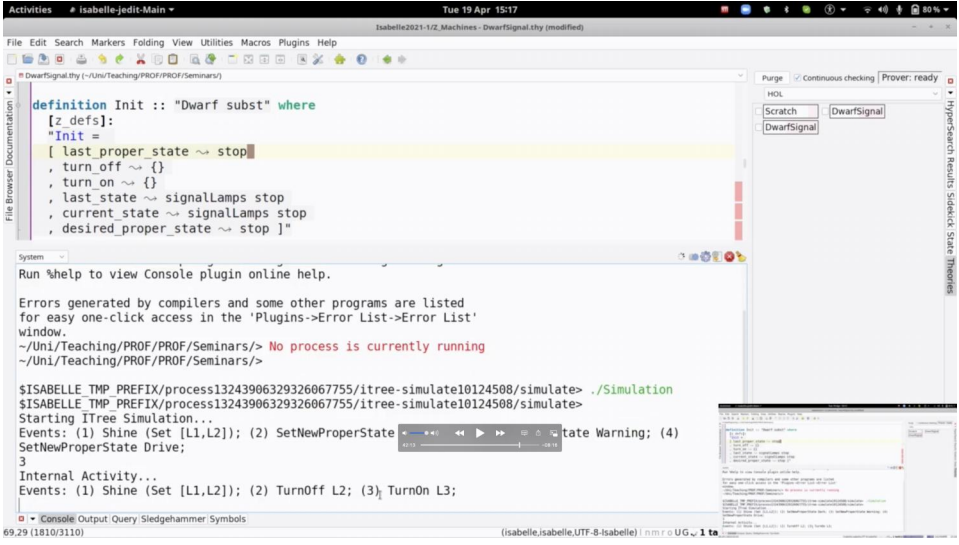
Checking the Specification



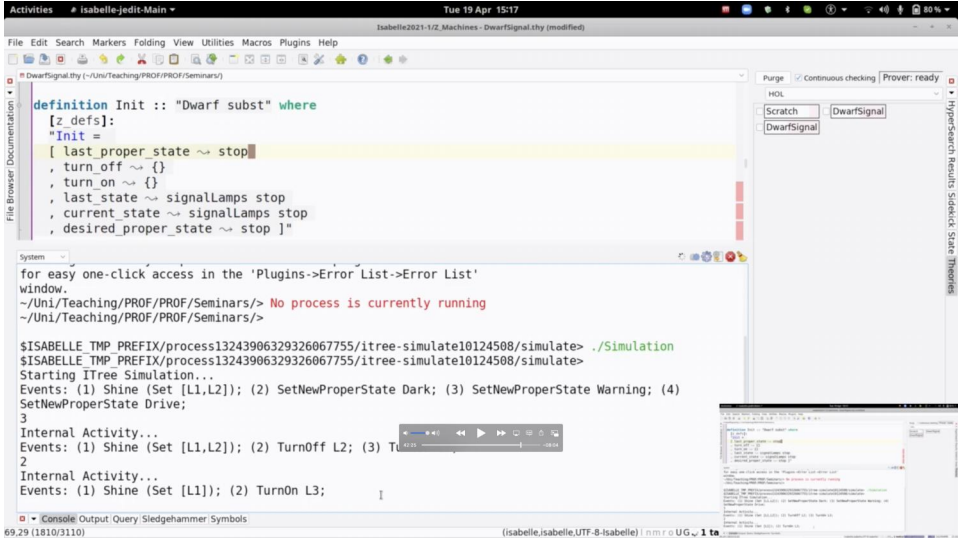
Checking the Specification



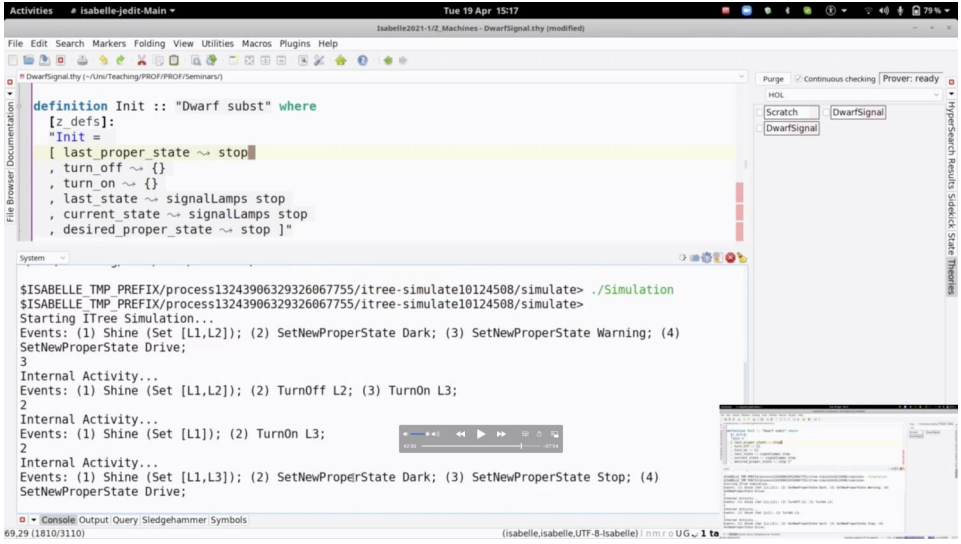
Checking the Specification



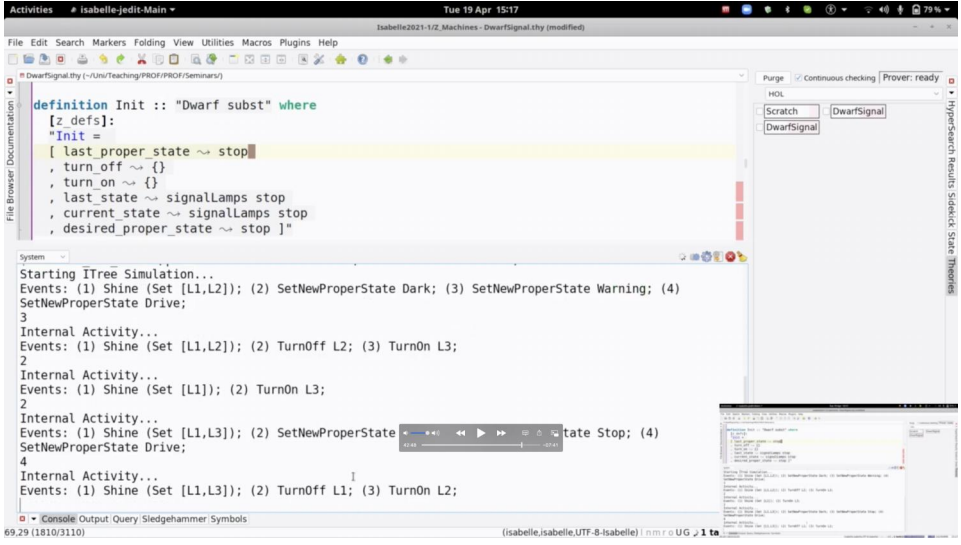
Checking the Specification



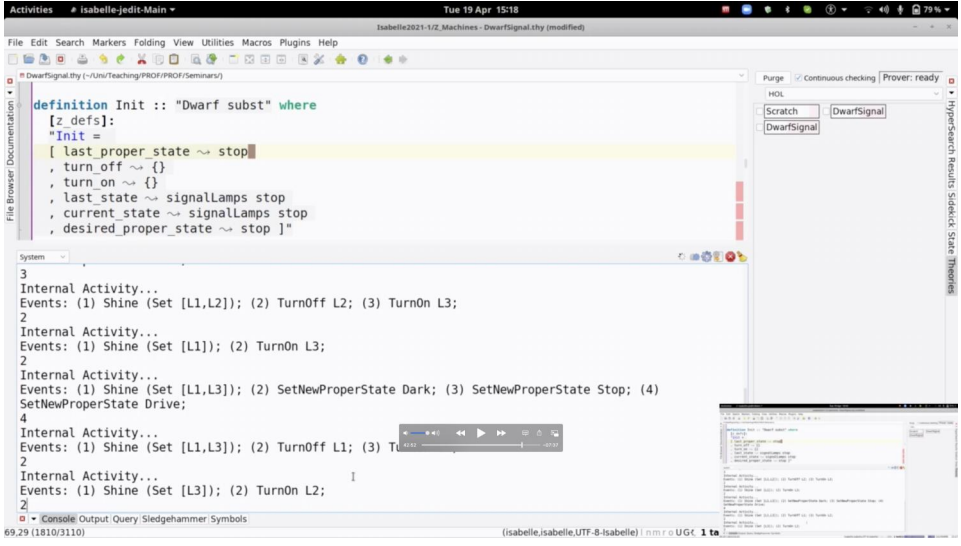
Checking the Specification



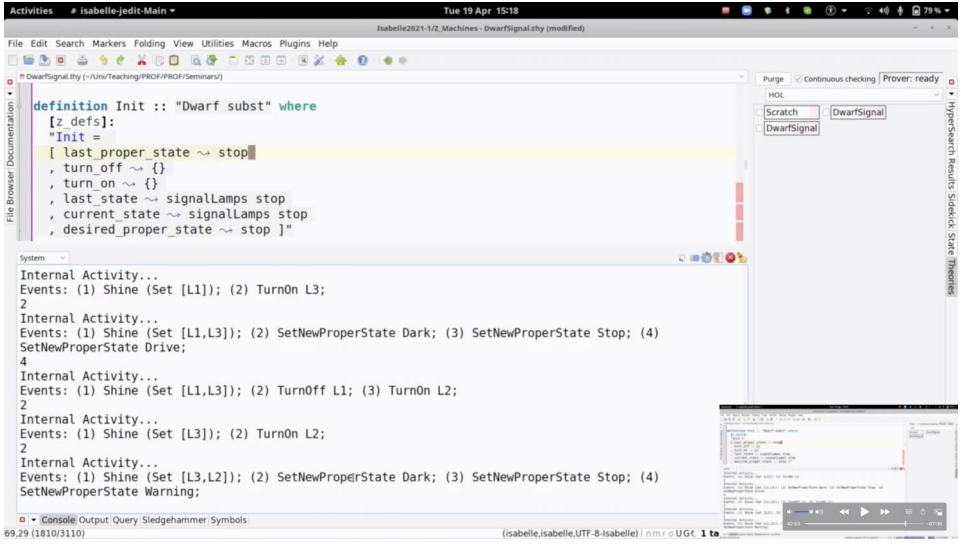
Checking the Specification



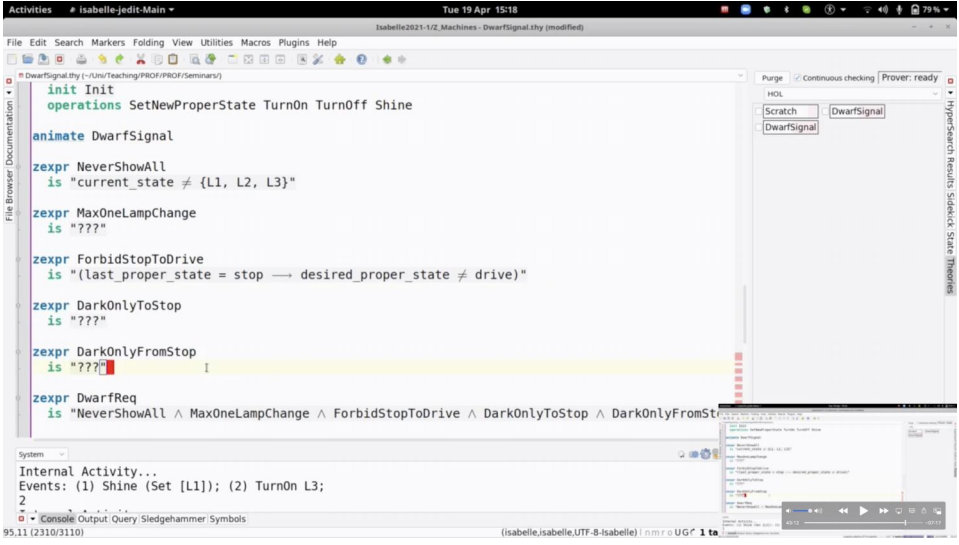
Checking the Specification



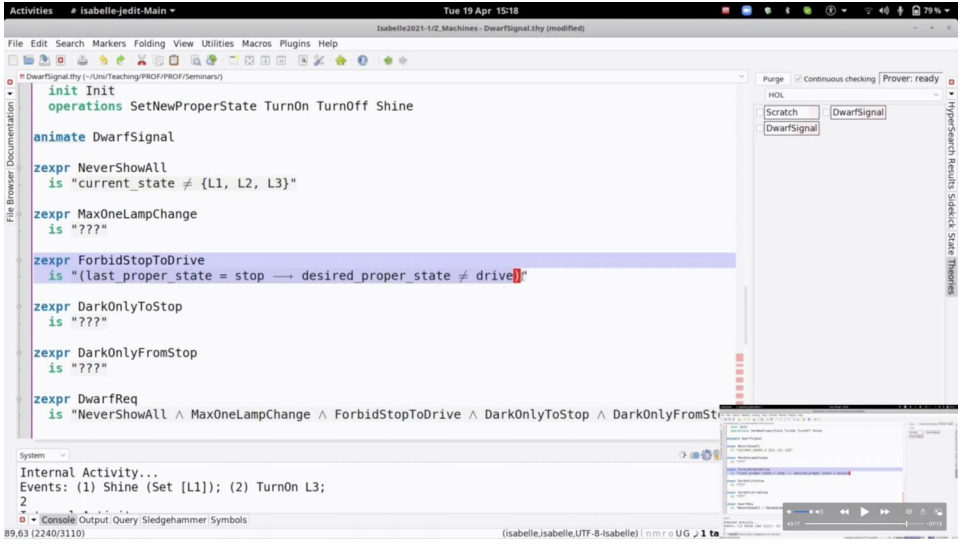
Checking the Specification



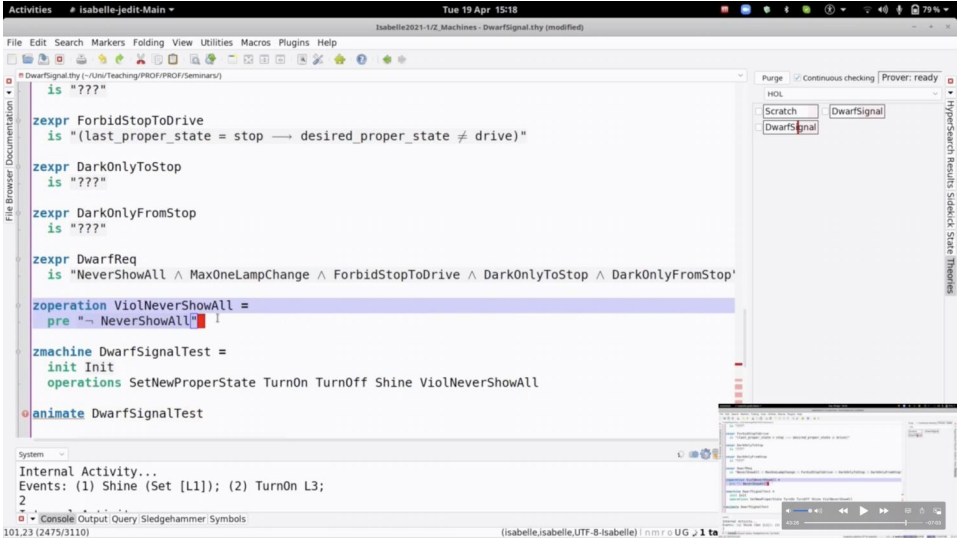
Checking the Specification



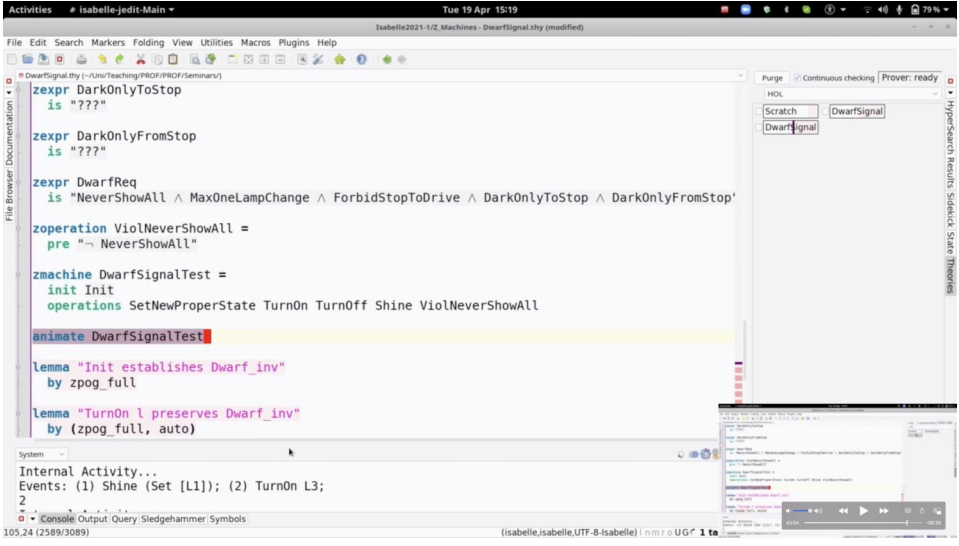
Checking the Specification



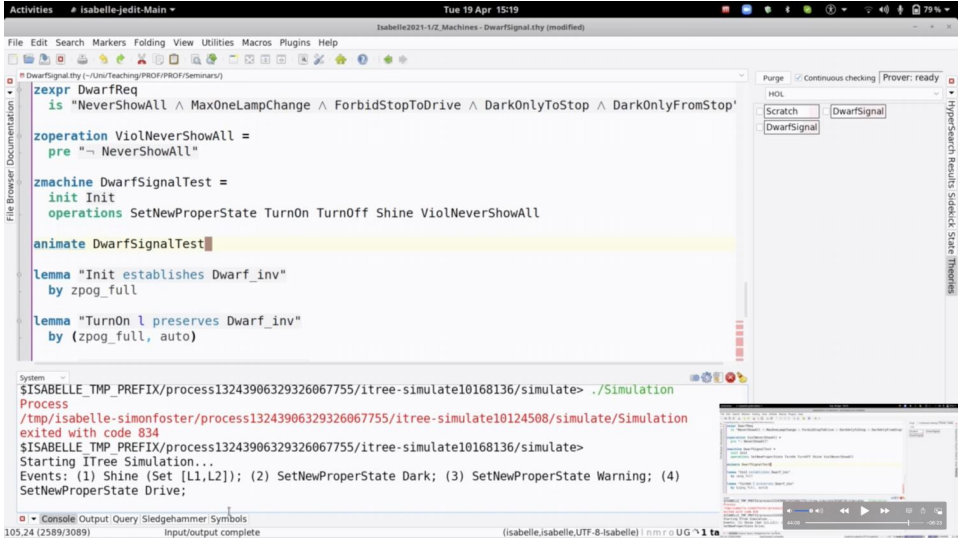
Checking the Specification



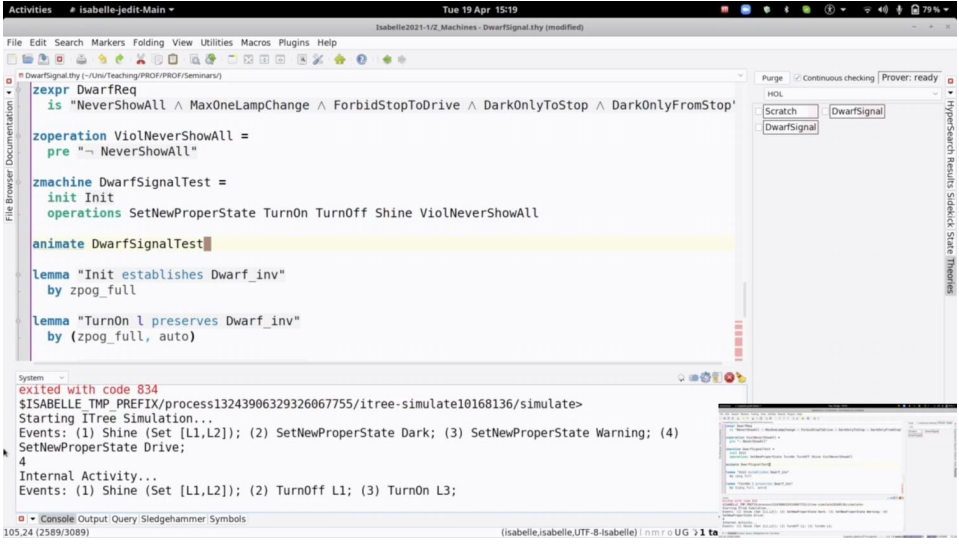
Checking the Specification



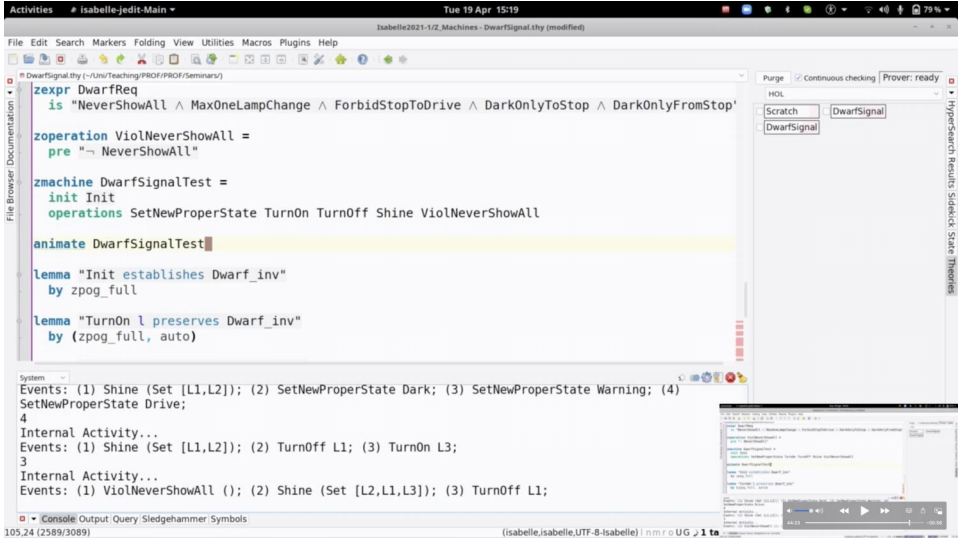
Checking the Specification



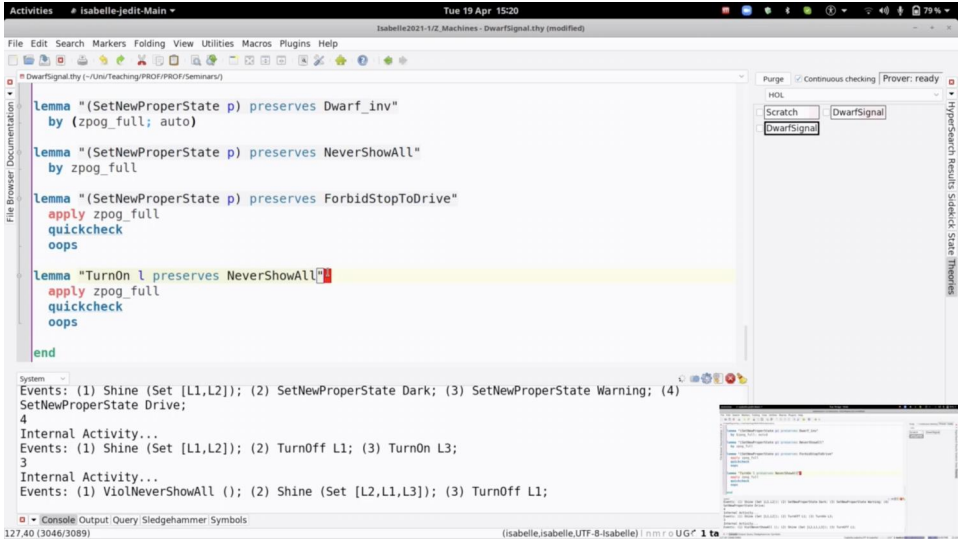
Checking the Specification



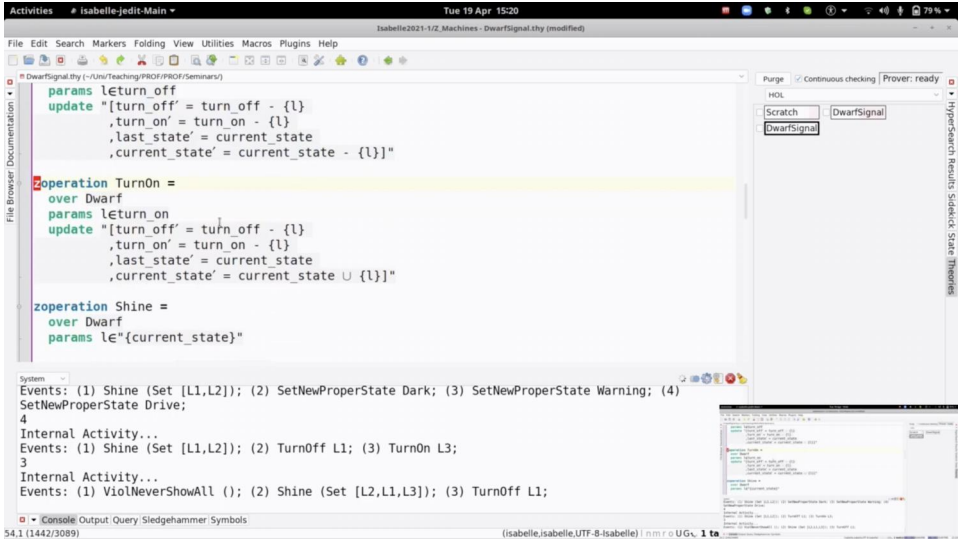
Checking the Specification



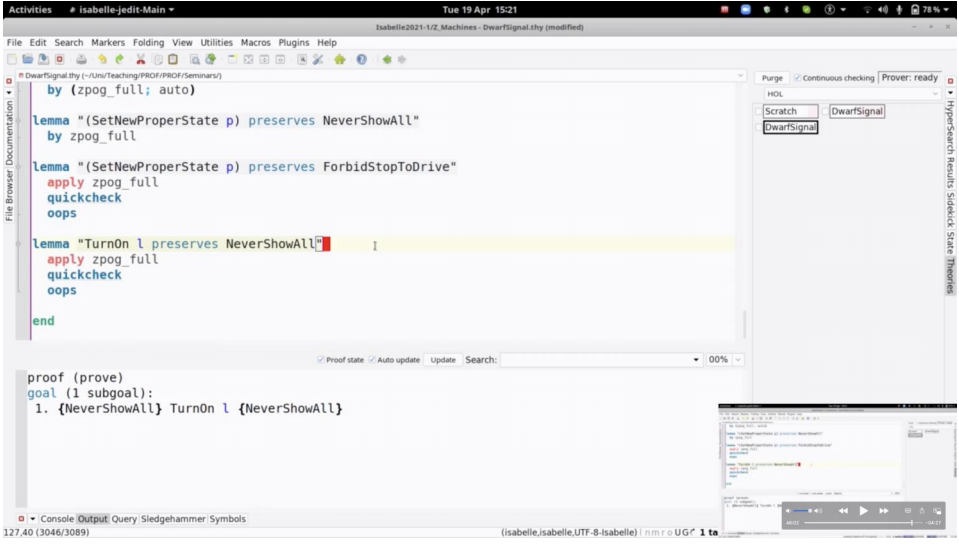
Checking the Specification



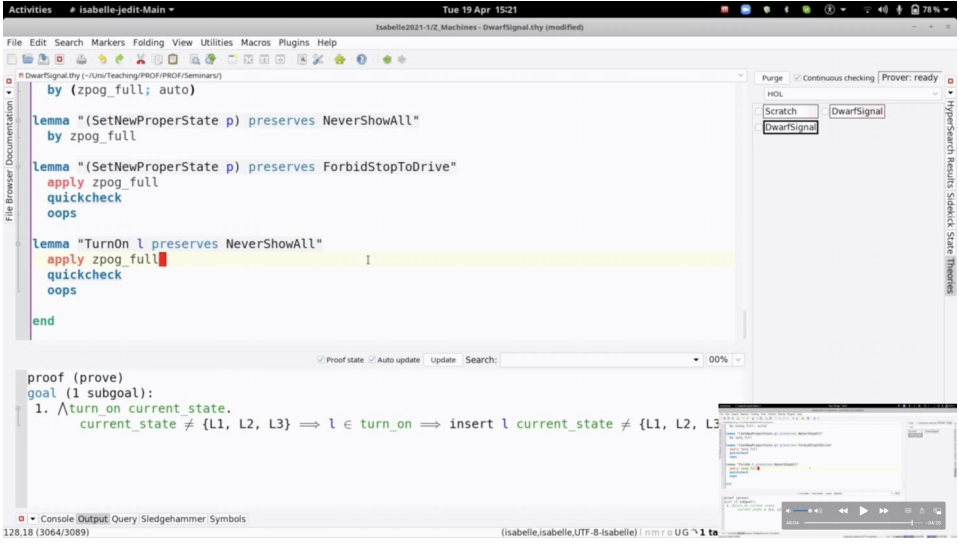
Checking the Specification



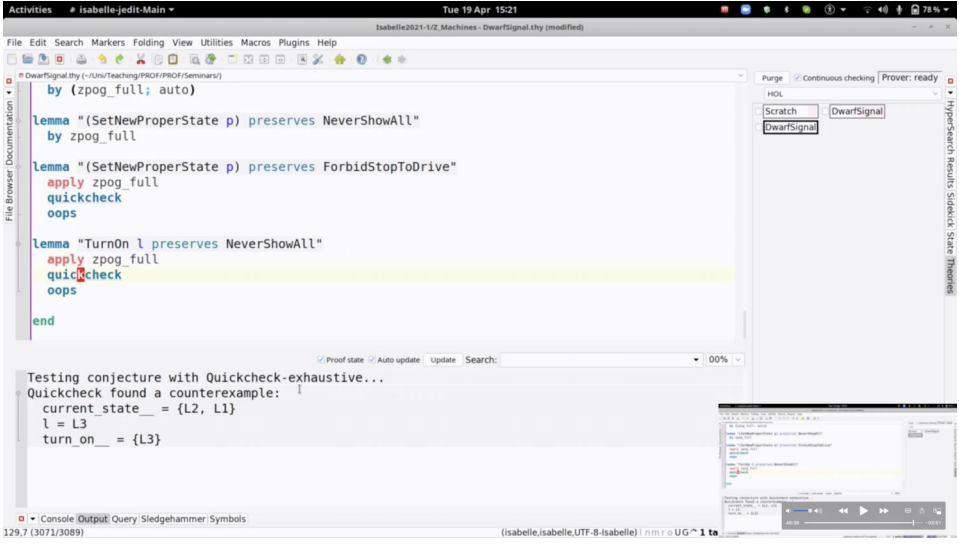
Checking the Specification



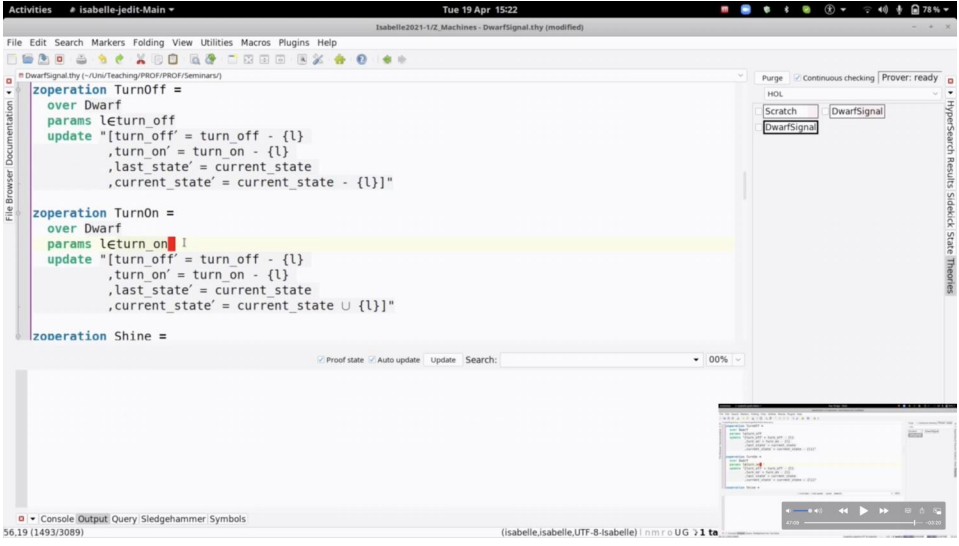
Checking the Specification



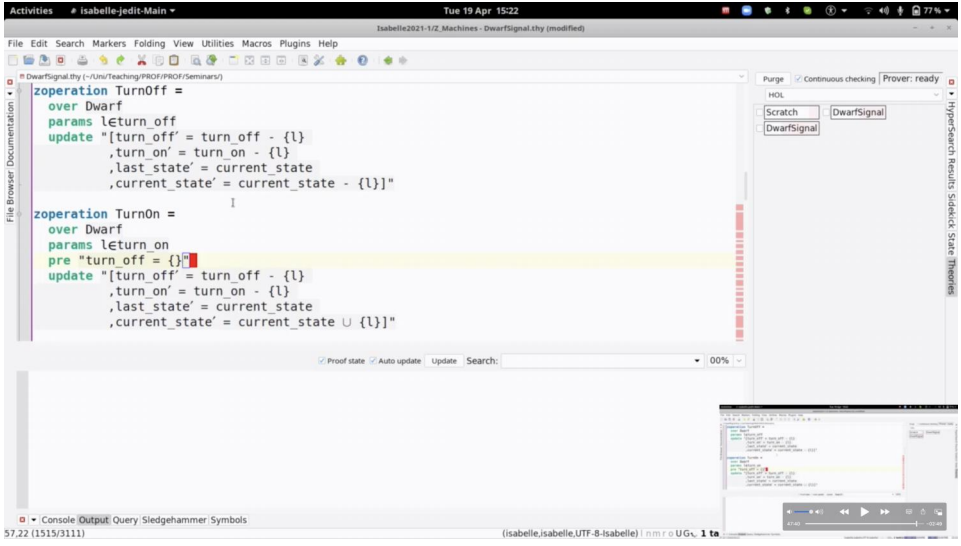
Checking the Specification



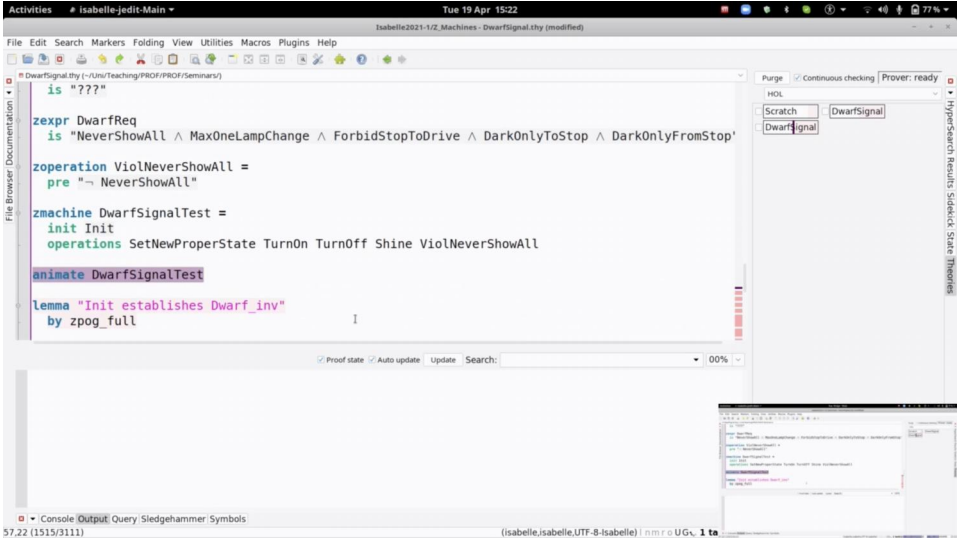
Checking the Specification



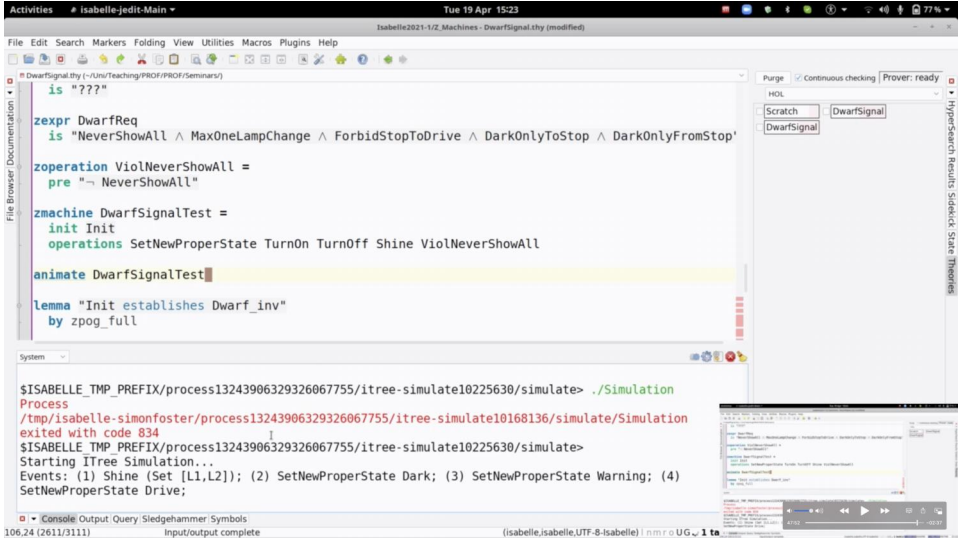
Checking the Specification



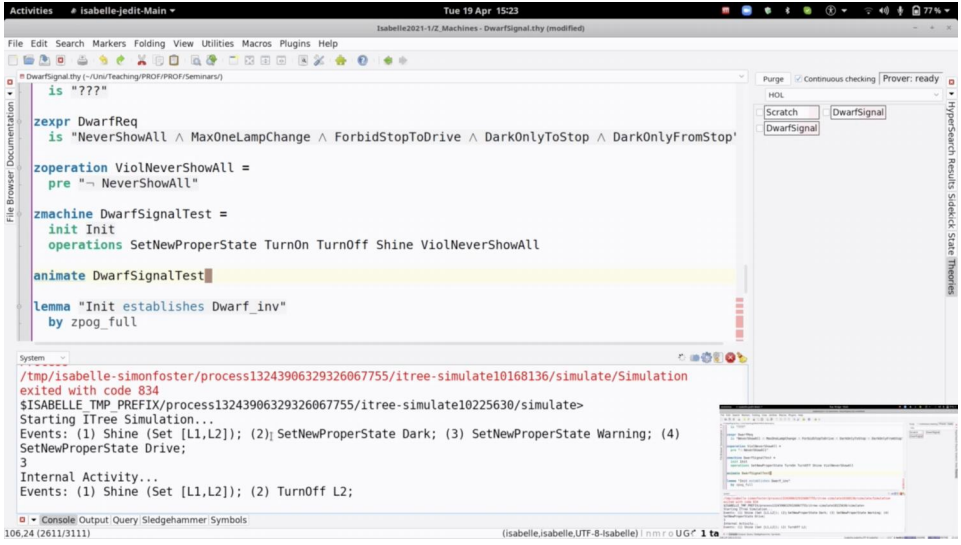
Checking the Specification



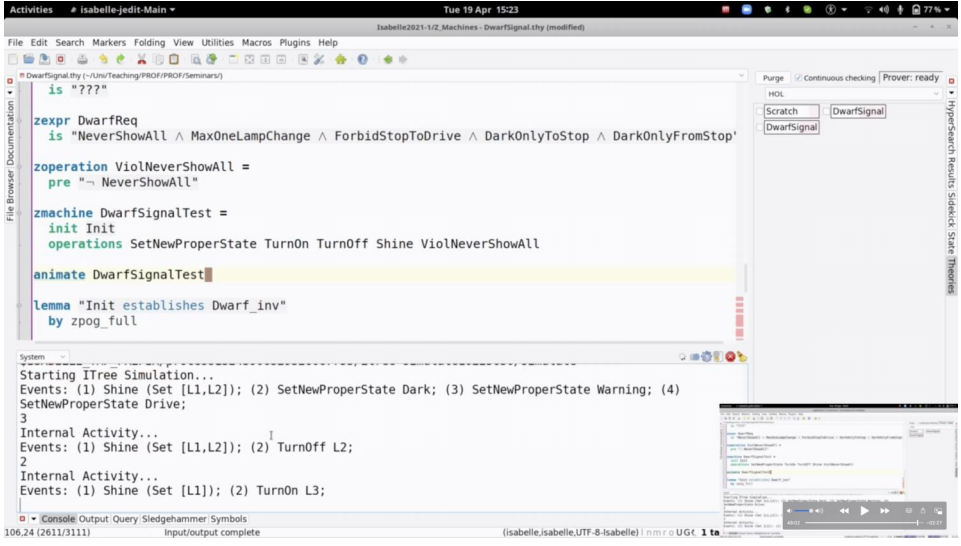
Checking the Specification



Checking the Specification



Checking the Specification



This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.

This lecture

- Modelling Z specifications with Z machines.
- Types, stores, and state variables.
- Operations and machines.
- Animation and verification.
- Managing requirements.