

■ 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

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



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:

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:

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:

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:

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:

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:

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:

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:

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**.

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



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  $\Rightarrow$  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  $\Rightarrow$  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 \ turn_off) \cup turn_on = desired_proper_state

turn_off \cap turn_on = \emptyset

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 \ turn_off) \cup turn_on = desired_proper_state

turn_off \cap turn_on = \emptyset

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) \cup turn_on = signalLamps desired_proper_state"

"turn_on \cap 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

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



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

Δ *DwarfSignal*

st? : *ProperState*

current_state = *desired_proper_state*

st? \neq *current_state*

last_proper_state' = *current_state*

turn_off' = *current_state* \ *st?*

turn_on' = *st?* \ *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

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



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:
 - The initial state assignment establishes the invariants.
 - 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:
 - The initial state assignment establishes the invariants.
 - 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:
 - The initial state assignment establishes the invariants.
 - 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

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



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  $\neq$  {L1, L2, L3}"
```

```
zexpr ForbidStopToDrive
```

```
  is "(last_proper_state = stop  $\longrightarrow$  desired_proper_state  
     $\neq$  drive)"
```

Example: Dwarf Signal Requirements

```
zexpr NeverShowAll
```

```
  is "current_state  $\neq$  {L1, L2, L3}"
```

```
zexpr ForbidStopToDrive
```

```
  is "(last_proper_state = stop  $\longrightarrow$  desired_proper_state  
       $\neq$  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  $\neq$  {L1, L2, L3}"
```

```
zexpr ForbidStopToDrive  
  is "(last_proper_state = stop  $\longrightarrow$  desired_proper_state  
     $\neq$  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

The screenshot shows the Isabelle/Isabelle IDE interface. The main window displays the theory file `DwarfSignal.thy` with the following content:

```
section < Dwarf Signal >

theory DwarfSignal
  imports "Z_Machines.Z_Machine"
begin

notation undefined ("???" )

subsection < State Space >

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  $\Rightarrow$  LampId set" where
  "signalLamps dark = {}" |
  "signalLamps stop = {L1, L2}" |
  "signalLamps warning = {L1, L3}" |
  "signalLamps drive = {L2, L3}"
```

The interface includes a menu bar (File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help), a toolbar, and a file browser on the left showing the file `DwarfSignal.thy`. The right sidebar shows the 'HyperSearch Results' and 'Sidekick' tabs, with 'DwarfSignal' selected. The bottom status bar shows the console output, the current file name `(isabelle.isabelle,UTF-8-Isabelle)`, and the page number `38,1 (875/3089)`.

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays the file DwarfSignal.thy with the following code:

```
update "[turn_off' = turn_off - {l}
,turn_on' = turn_on - {l}
,last_state' = current_state
,current_state' = current_state - {l}]"

zoperation TurnOn =
over Dwarf
params l return_on
update "[turn_off' = turn_off - {l}
,turn_on' = turn_on - {l}
,last_state' = current_state
,current_state' = current_state U {l}]"

zoperation Shine =
over Dwarf
params l e "{current_state}"

definition Init :: "Dwarf subst" where
[z_defs]:
"Init =
[ last_proper_state ~ stop
, turn_off ~ {}
, turn_on ~ {}
, last_state ~ signalLamps stop
```

On the right side, the HyperSearch interface is visible, showing a search for 'DwarfSignal' with results for 'Scratch' and 'DwarfSignal'. The 'DwarfSignal' result is selected.

At the bottom, the console shows the output of the Sledgehammer prover, indicating that the proof was successful. The status bar at the bottom left shows '62,1 (1658/3089)' and the bottom right shows '(isabelle,isabelle,UTF-8-Isabelle) | n m r'.

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays the DwarfSignal.thy file with the following code:

```
operation Shine =  
  over Dwarf  
  params le"{current_state}"  
  
definition Init :: "Dwarf subst" where  
  [z_defs]:  
  "Init =  
  [ last_proper_state ~ stop  
  , turn_off ~ {}  
  , turn_on ~ {}  
  , last_state ~ signalLamps stop  
  , current_state ~ signalLamps stop  
  , desired_proper_state ~ stop]"  
  
zmachine DwarfSignal =  
  init Init  
  operations SetNewProperState TurnOn TurnOff Shine  
  
zexpr NeverShowAll  
  is "current_state ≠ {L1, L2, L3}"  
  
zexpr MaxOneLampChange  
  is "???"
```

The right sidebar shows the HyperSearch Results, Sidekick, and State Theories. The State Theories section is expanded to show the DwarfSignal theory.

The bottom status bar shows the proof state: (isabelle.isabelle.UTF-8-Isabelle) | n m r

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays the content of `DwarfSignal.thy`. The code defines an initial state `Init` and a machine `DwarfSignal` with various operations and an animation function. A `zexpr` `NeverShowAll` is also defined. The right-hand side of the editor shows the prover interface, including a `Purge` button, a `Continuous checking` checkbox, and a `Prover: ready` status. Below the prover interface is a `HyperSearch Results` sidebar. At the bottom of the editor, there is a `Proof state` checkbox, an `Auto update` checkbox, an `Update` button, and a `Search:` field. A video player control bar is visible at the bottom center of the editor window. The bottom status bar shows the console output, the current theory `(isabelle.isabelle.UTF-8-Isabelle)`, and the file name `nmr`.

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"

zmachine DwarfSignal =
  init Init
  operations SetNewProperState TurnOn TurnOff Shine

animate

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

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays the content of `DwarfSignal.thy`. The code defines an initial state `Init` and a state machine `DwarfSignal`. The `animate` command is highlighted in yellow. A console window at the bottom shows the output of the `animate` command, indicating that the animation is compiling.

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"

zmachine DwarfSignal =
  init Init
  operations SetNewProperState TurnOn TurnOff Shine

animate DwarfSignal

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

See theory exports
Compiling animation...

Console Output Query Sledgehammer Symbols
80,20 (2061/3110) Input/output complete (isabelle,isabelle,UTF-8-Isabelle) | nmr o UG

Checking the Specification

The screenshot displays the Isabelle/Proof General environment. The main window shows the source file `DwarfSignal.thy` with the following code:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"

zmachine DwarfSignal =
  init Init
  operations SetNewProperState TurnOn TurnOff Shine
```

Below the code editor, the console shows the following output:

```
See theory exports
Compiling animation...
See theory exports
Start animation
```

The interface also includes a file browser on the left, a search bar at the bottom, and a status bar at the very bottom showing the session ID `80,20 (2061/3110)` and the current file `(isabelle,isabelle,UTF-8-Isabelle) | nmr o UG`.

Checking the Specification

The screenshot shows the Isabelle2021-1/2_Machines - DwarfSignal.thy (modified) window. The main editor displays the following code:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"

zmachine DwarfSignal =
  init Init
  operations SetNewProperState TurnOn TurnOff Shine
```

Below the code, the console output shows:

```
See theory exports
Compiling animation...
See theory exports
Start animation
```

The console also shows a progress bar and a search field. The status bar at the bottom indicates the file path and the user's name: (isabelle,isabelle,UTF-8-Isabelle) | n.m.r.o UG

Checking the Specification

The screenshot shows the Isabelle2021-1/2_Machines - DwarfSignal.thy (modified) window. The main editor displays the following definition:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"
```

The console window shows the following output:

```
Press %<TAB> to list built-in commands.
Run built-in with --help argument to get a brief usage message.
Run %help to view Console plugin online help.

Errors generated by compilers and some other programs are listed
for easy one-click access in the 'Plugins->Error List->Error List'
window.
~/Uni/Teaching/PROF/PROF/Seminars/> No process is currently running
~/Uni/Teaching/PROF/PROF/Seminars/>

$ISABELLE_TMP_PREFIX/process13243906329326067755/itree te> ./Simulation
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree te>
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;
```

The right sidebar shows the HyperSearch Results Sidekick State Theories window with the following entries:

- HOL
- Scratch
- DwarfSignal

The bottom status bar shows: 80,20 (2061/3110) Input/output complete (isabelle,isabelle.UTF-8-Isabelle) | nmr o UG 1 ta

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays a theorem definition:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"
```

The right sidebar shows the 'HyperSearch Results' and 'Sidekick State Theories' panels. The 'Sidekick State Theories' panel lists 'Scratch' and 'DwarfSignal'.

The console window shows the following output:

```
Run %help to view Console plugin online help.

Errors generated by compilers and some other programs are listed
for easy one-click access in the 'Plugins->Error List->Error List'
window.
~/Uni/Teaching/PROF/PROF/Seminars/> No process is currently running
~/Uni/Teaching/PROF/PROF/Seminars/>

$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10124508/simulate> ./Simulation
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10124508/simulate>
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState
SetNewProperState Drive;
3
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L2; (3) TurnOn L3;
```

The bottom status bar shows the file path: (isabelle.isabelle.UTF-8-Isabelle) | in m r o U G w 1 ta

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays a definition for 'Init' in a 'Dwarf subst' theory. The definition is as follows:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"
```

The right-hand pane shows the 'HyperSearch Results' for the 'DwarfSignal' theory, with 'DwarfSignal' selected. Below the editor is a console window showing the execution of a simulation:

```
for easy one-click access in the 'Plugins->Error List->Error List'
window.
~/Uni/Teaching/PROF/PROF/Seminars/> No process is currently running
~/Uni/Teaching/PROF/PROF/Seminars/>

$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10124508/simulate> ./Simulation
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10124508/simulate>
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;
3
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L2; (3) T
2
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
```

The console output is accompanied by a small video player interface with a play button and a progress bar.

Checking the Specification

The screenshot shows the Isabelle/Coq IDE interface. The main window displays a theorem definition:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"
```

The right sidebar shows the "HyperSearch Results: Sidekick State Theories" panel with "DwarfSignal" selected. Below the editor is a "System" console window showing the execution of a simulation:

```
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10124508/simulate> ./Simulation
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10124508/simulate>
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;
3
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L2; (3) TurnOn L3;
2
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
2
Internal Activity...
Events: (1) Shine (Set [L1,L3]); (2) SetNewProperState Dark; (3) SetNewProperState Stop; (4)
SetNewProperState Drive;
```

A playback control bar is visible over the console output, indicating a duration of 02:33 to 07:34. The bottom status bar shows the file path and the user's name: "(isabelle,isabelle.UTF-8-Isabelle) | n m r o U G | 1 ta".

Checking the Specification

The screenshot displays the Isabelle/PTT IDE interface. At the top, the window title is "Isabelle2021-1/2_Machines - DwarfSignal.thy (modified)". The menu bar includes "File", "Edit", "Search", "Markers", "Folding", "View", "Utilities", "Macros", "Plugins", and "Help".

The main editor shows the following code snippet:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"
```

The line `[last_proper_state ~ stop` is highlighted in yellow. To the right of the editor, there is a "Purge" button, a checked "Continuous checking" checkbox, and a "Prover: ready" status. Below this, a "HOL" section contains two checkboxes, both labeled "DwarfSignal".

The console window at the bottom shows the following output:

```
System
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;
3
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L2; (3) TurnOn L3;
2
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
2
Internal Activity...
Events: (1) Shine (Set [L1,L3]); (2) SetNewProperState
SetNewProperState Drive;
4
Internal Activity...
Events: (1) Shine (Set [L1,L3]); (2) TurnOff L1; (3) TurnOn L2;
```

A playback control bar is visible over the console output, showing a progress bar from 02:03 to 07:41.

At the bottom left, the status bar shows "69,29 (1810/3110)". At the bottom right, it shows "(isabelle,isabelle.UTF-8-Isabelle) | n m r o U G | 1 ta".

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays a definition for `Init` in the `DwarfSignal` theory:

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"
```

The right-hand pane shows the `HyperSearch Results` for the `DwarfSignal` theory, with `Scratch` and `DwarfSignal` theories listed.

The bottom pane shows the execution trace for the `System` component:

```
3
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L2; (3) TurnOn L3;
2
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
2
Internal Activity...
Events: (1) Shine (Set [L1,L3]); (2) SetNewProperState Dark; (3) SetNewProperState Stop; (4)
SetNewProperState Drive;
4
Internal Activity...
Events: (1) Shine (Set [L1,L3]); (2) TurnOff L1; (3) T
2
Internal Activity...
Events: (1) Shine (Set [L3]); (2) TurnOn L2;
2
```

The bottom status bar shows the console output: `69.29 (1810/3110) (isabelle,isabelle,UTF-8-Isabelle) | n m r o U G C 1 t a`.

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays a definition for `Init` within a `"Dwarf subst"` context. The definition includes a `z_defs` block and an `Init` function that returns a list of state transitions. The `last_proper_state` transition is highlighted in yellow.

```
definition Init :: "Dwarf subst" where
  [z_defs]:
  "Init =
  [ last_proper_state ~ stop
  , turn_off ~ {}
  , turn_on ~ {}
  , last_state ~ signalLamps stop
  , current_state ~ signalLamps stop
  , desired_proper_state ~ stop ]"
```

Below the editor, the console output shows the execution trace of the `Init` function. It consists of four "Internal Activity..." blocks, each followed by a list of events. The events include `Shine`, `TurnOn`, `SetNewProperState`, and `Warning`.

```
System
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
2
Internal Activity...
Events: (1) Shine (Set [L1,L3]); (2) SetNewProperState Dark; (3) SetNewProperState Stop; (4)
SetNewProperState Drive;
4
Internal Activity...
Events: (1) Shine (Set [L1,L3]); (2) TurnOff L1; (3) TurnOn L2;
2
Internal Activity...
Events: (1) Shine (Set [L3]); (2) TurnOn L2;
2
Internal Activity...
Events: (1) Shine (Set [L3,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Stop; (4)
SetNewProperState Warning;
```

The right sidebar shows the `HyperSearch Results` and `Sidekick State Theories` panels. The `Sidekick State Theories` panel lists `Scratch` and `DwarfSignal`. The `Console Output` panel at the bottom shows the execution trace.

Checking the Specification

The screenshot shows the Isabelle/Thy editor interface. The main window displays the code for `DwarfSignal.thy`. The code includes an `init` block for `Init` with operations `SetNewProperState`, `TurnOn`, `TurnOff`, and `Shine`. It also defines an `animate` block for `DwarfSignal` and several `zexpr` (Z3 expressions) for properties like `NeverShowAll`, `MaxOneLampChange`, `ForbidStopToDrive`, `DarkOnlyToStop`, `DarkOnlyFromStop`, and `DwarfReq`. The `DwarfReq` property is defined as a conjunction of the other properties.

```
init Init
operations SetNewProperState TurnOn TurnOff Shine

animate DwarfSignal

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

zexpr MaxOneLampChange
is "???"

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

zexpr DarkOnlyToStop
is "???"

zexpr DarkOnlyFromStop
is "???"

zexpr DwarfReq
is "NeverShowAll ∧ MaxOneLampChange ∧ ForbidStopToDrive ∧ DarkOnlyToStop ∧ DarkOnlyFromStop"
```

The console output at the bottom shows the system's internal activity, including events like `Shine (Set [L1])` and `TurnOn L3`.

System
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
2

Console Output Query Sledgehammer Symbols

95,11 (2310/3110) (isabelle.isabelle.UTF-8-Isabelle) | n m r o U G 1 ta

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays the DwarfSignal.thy file with the following code:

```
init Init
operations SetNewProperState TurnOn TurnOff Shine

animate DwarfSignal

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

zexpr MaxOneLampChange
is "???"

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

zexpr DarkOnlyToStop
is "???"

zexpr DarkOnlyFromStop
is "???"

zexpr DwarfReq
is "NeverShowAll ∧ MaxOneLampChange ∧ ForbidStopToDrive ∧ DarkOnlyToStop ∧ DarkOnlyFromStop"
```

The console output at the bottom shows the system's internal activity:

```
System
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
2
```

The right sidebar shows the HyperSearch Results for DwarfSignal, with the DwarfSignal theory selected. The bottom status bar indicates the file path: (isabelle.isabelle.UTF-8-Isabelle) in m r o U G 1 ta.

Checking the Specification

The screenshot shows a theorem prover interface with the following components:

- Header:** Activities, Isabelle-jedit-Main, Tue 19 Apr 15:18, Isabelle2021-1/2_Machines - DwarfSignal.thy (modified)
- Menu:** File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help
- File Browser:** DwarfSignal.thy (~/Uni/Teaching/PROF/PROF/Seminars/)
- Main Editor:**

```
is "???"  
  
zexpr ForbidStopToDrive  
is "(last_proper_state = stop  $\longrightarrow$  desired_proper_state  $\neq$  drive)"  
  
zexpr DarkOnlyToStop  
is "???"  
  
zexpr DarkOnlyFromStop  
is "???"  
  
zexpr DwarfReq  
is "NeverShowAll  $\wedge$  MaxOneLampChange  $\wedge$  ForbidStopToDrive  $\wedge$  DarkOnlyToStop  $\wedge$  DarkOnlyFromStop"  
  
zoperation ViolNeverShowAll =  
pre " $\neg$  NeverShowAll"  
  
zmachine DwarfSignalTest =  
  init Init  
  operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll  
  
animate DwarfSignalTest
```
- Right Panel:** Purge, Continuous checking, Prover: ready, HOL, Scratch, DwarfSignal, DwarfSignal
- Bottom Console:**

```
System  
Internal Activity...  
Events: (1) Shine (Set [L1]); (2) TurnOn L3;  
2
```
- Bottom Status Bar:** 101,23 (2475/3110) (isabelle,isabelle,UTF-8-Isabelle) | n m r o U G | 1 ta

Checking the Specification

Activities # isabelle-jedit-Main Tue 19 Apr 15:19
Isabelle2021-1/2_Machines - DwarfSignal.thy (modified)

File Edit Search Markers Folding View Utilities Macros Plugins Help

DwarfSignal.thy (~/Uni/Teaching/PROF/PROF/Seminars)

```
zexpr DarkOnlyToStop
is "???"

zexpr DarkOnlyFromStop
is "???"

zexpr DwarfReq
is "NeverShowAll  $\wedge$  MaxOneLampChange  $\wedge$  ForbidStopToDrive  $\wedge$  DarkOnlyToStop  $\wedge$  DarkOnlyFromStop"

zoperation ViolNeverShowAll =
pre " $\neg$  NeverShowAll"

zmachine DwarfSignalTest =
init Init
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll

animate DwarfSignalTest

lemma "Init establishes Dwarf_inv"
by z Pog_full

lemma "TurnOn l preserves Dwarf_inv"
by (z Pog_full, auto)
```

System
Internal Activity...
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
2

Console Output Query Sledgehammer Symbols

105.24 (2589/3089) (isabelle.isabelle.UTF-8-Isabelle) | n m r o U G 1 ta

HyperSearch Results Sidekick State Theories

Purge Continuous checking Prover: ready
HOL
Scratch DwarfSignal
DwarfSignal

Checking the Specification

The screenshot shows the Isabelle2021-1/2_Machines - DwarfSignal.thy (modified) file in a text editor. The code defines a DwarfSignal machine and its simulation. The console output shows the simulation results.

```
zexpr DwarfReq
is "NeverShowAll ^ MaxOneLampChange ^ ForbidStopToDrive ^ DarkOnlyToStop ^ DarkOnlyFromStop"

zoperation ViolNeverShowAll =
pre "¬ NeverShowAll"

zmachine DwarfSignalTest =
init Init
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll

animate DwarfSignalTest

lemma "Init establishes Dwarf_inv"
by zpog_full

lemma "TurnOn l preserves Dwarf_inv"
by (zpog_full, auto)
```

System
\$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10168136/simulate> ./Simulation
Process
/tmp/isabelle-simonfoster/process13243906329326067755/itree-simulate10124508/simulate/Simulation
exited with code 834
\$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10168136/simulate>
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;

Checking the Specification

The screenshot shows the Isabelle/Isabelle IDE interface. The main editor window displays the following code for `DwarfSignal.thy`:

```
zexpr DwarfReq
is "NeverShowAll  $\wedge$  MaxOneLampChange  $\wedge$  ForbidStopToDrive  $\wedge$  DarkOnlyToStop  $\wedge$  DarkOnlyFromStop"

zoperation ViolNeverShowAll =
pre " $\neg$  NeverShowAll"

zmachine DwarfSignalTest =
init Init
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll

animate DwarfSignalTest

lemma "Init establishes Dwarf_inv"
by zpog_full

lemma "TurnOn l preserves Dwarf_inv"
by (zpog_full, auto)
```

The console window at the bottom shows the execution of a simulation:

```
System
exited with code 834
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10168136/simulate>
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;
4
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L1; (3) TurnOn L3;
```

The right sidebar shows the HyperSearch Results, Sidekick, and State Theories. The State Theories panel shows the current state of the simulation, including the DwarfSignal machine and its state.

Checking the Specification

The screenshot displays the Isabelle/TPIC environment. The main window shows the file DwarfSignal.thy with the following content:

```
zexpr DwarfReq
is "NeverShowAll  $\wedge$  MaxOneLampChange  $\wedge$  ForbidStopToDrive  $\wedge$  DarkOnlyToStop  $\wedge$  DarkOnlyFromStop"

zoperation ViolNeverShowAll =
pre " $\neg$  NeverShowAll"

zmachine DwarfSignalTest =
init Init
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll

animate DwarfSignalTest

lemma "Init establishes Dwarf_inv"
by zpog_full

lemma "TurnOn l preserves Dwarf_inv"
by (zpog_full, auto)
```

The console window at the bottom shows the following execution trace:

```
System
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;
4
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L1; (3) TurnOn L3;
3
Internal Activity...
Events: (1) ViolNeverShowAll (); (2) Shine (Set [L2,L1,L3]); (3) TurnOff L1;
```

On the right side, the HyperSearch Results panel shows a search for 'DwarfSignal' with two results listed.

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays a theorem script with several lemmas. The fourth lemma is highlighted in yellow:

```
lemma "TurnOn l preserves NeverShowAll"  
  apply z Pog_full  
  quickcheck  
  oops  
end
```

On the right side, the "HyperSearch Results: Sidekick State Theories" panel shows a tree structure with "DwarfSignal" selected.

The console output at the bottom shows the following sequence of events and internal activities:

```
System  
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)  
SetNewProperState Drive;  
4  
Internal Activity...  
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L1; (3) TurnOn L3;  
3  
Internal Activity...  
Events: (1) ViolNeverShowAll (); (2) Shine (Set [L2,L1,L3]); (3) TurnOff L1;
```

The status bar at the bottom indicates the file path: (isabelle,isabelle,UTF-8-Isabelle) | in m r o U G | 1 ta

Checking the Specification

The screenshot shows the Isabelle/Thy editor interface. The main window displays the code for `DwarfSignal.thy`, which defines parameters, update functions, and operations for a Dwarf system. The console at the bottom shows the execution of a proof, including system events and internal activities.

```
params leturn_off
update "[turn_off' = turn_off - {l}
,turn_on' = turn_on - {l}
,last_state' = current_state
,current_state' = current_state - {l}]"

zoperation TurnOn =
over Dwarf
params leturn_on
update "[turn_off' = turn_off - {l}
,turn_on' = turn_on - {l}
,last_state' = current_state
,current_state' = current_state  $\cup$  {l}]"

zoperation Shine =
over Dwarf
params letcurrent_state"
```

System
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4) SetNewProperState Drive;
4
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L1; (3) TurnOn L3;
3
Internal Activity...
Events: (1) ViolNeverShowAll (); (2) Shine (Set [L2,L1,L3]); (3) TurnOff L1;

54,1 (1442/3089) (isabelle,isabelle.UTF-8-Isabelle) | n m r o U G v. 1 t a

Checking the Specification

The screenshot shows the Isabelle/Isabelle IDE interface. The main editor displays the following code:

```
by (zpgog_full; auto)

lemma "(SetNewProperState p) preserves NeverShowAll"
  by zpgog_full

lemma "(SetNewProperState p) preserves ForbidStopToDrive"
  apply zpgog_full
  quickcheck
  oops

lemma "TurnOn l preserves NeverShowAll"
  apply zpgog_full
  quickcheck
  oops
end
```

Below the code, a proof goal is shown:

```
proof (prove)
goal (1 subgoal):
1. {NeverShowAll} TurnOn l {NeverShowAll}
```

The IDE interface includes a menu bar (File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help), a toolbar, and a status bar at the bottom showing the file path and the current position in the document.

Checking the Specification

The screenshot shows the Isabelle/HOL IDE interface. The main editor displays the following code:

```
by (zpgog_full; auto)

lemma "(SetNewProperState p) preserves NeverShowAll"
  by zpgog_full

lemma "(SetNewProperState p) preserves ForbidStopToDrive"
  apply zpgog_full
  quickcheck
  oops

lemma "TurnOn l preserves NeverShowAll"
  apply zpgog_full
  quickcheck
  oops
end

proof (prove)
goal (1 subgoal):
1.  $\bigwedge \text{turn\_on\_current\_state. current\_state} \neq \{L1, L2, L3\} \implies l \in \text{turn\_on} \implies \text{insert } l \text{ current\_state} \neq \{L1, L2, L3\}$ 
end
```

The IDE interface includes a top menu bar (File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help), a toolbar, and a status bar at the bottom showing the file path and system information. On the right side, there is a 'HyperSearch Results: Sidekick State Theories' panel with a 'DwarfSignal' entry selected. A smaller window in the bottom right corner shows a detailed view of the proof progress, with a progress bar and a play button.

Checking the Specification

The screenshot shows the Isabelle/Thy editor interface. The main window displays a theorem proof for `TurnOn l preserves NeverShowAll`. The proof is structured as follows:

```
by (zpog_full; auto)

lemma "(SetNewProperState p) preserves NeverShowAll"
  by zpog_full

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

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

Below the proof, the console output shows the result of a quickcheck test:

```
Testing conjecture with Quickcheck-exhaustive...
Quickcheck found a counterexample: 1
  current_state__ = {L2, L1}
    l = L3
  turn_on__ = {L3}
```

The right sidebar shows the "HyperSearch Results: Sidekick State Theories" panel, with "DwarfSignal" selected. The bottom status bar indicates the file path `(isabelle,isabelle,UTF-8-Isabelle) | n m r o U G ^ 1 t a`.

Checking the Specification

The screenshot shows the Isabelle/TPP interface with the file `DwarfSignal.thy` open. The main editor displays the following code:

```
zoperation TurnOff =  
  over Dwarf  
  params leturn_off  
  update "[turn_off' = turn_off - {l}  
    ,turn_on' = turn_on - {l}  
    ,last_state' = current_state  
    ,current_state' = current_state - {l}]"  
  
zoperation TurnOn =  
  over Dwarf  
  params leturn_on |  
  update "[turn_off' = turn_off - {l}  
    ,turn_on' = turn_on - {l}  
    ,last_state' = current_state  
    ,current_state' = current_state  $\cup$  {l}]"  
  
zoperation Shine =
```

The `params leturn_on |` line in the `TurnOn` operation is highlighted in yellow. The interface includes a menu bar (File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help), a toolbar, and a file browser on the left. On the right, there is a prover status panel with buttons for "Purge", "Continuous checking", and "Prover: ready". Below these are checkboxes for "HOL", "Scratch", and "DwarfSignal". A vertical sidebar on the far right is labeled "HyperSearch Results: Sidekick State Theories". At the bottom, there is a console/output area and a status bar showing the file path and a timestamp.

Checking the Specification

The screenshot shows the Isabelle/Isabelle IDE interface. The main window displays the file `DwarfSignal.thy` with the following content:

```
zoperation TurnOff =  
  over Dwarf  
  params leturn_off  
  update "[turn_off' = turn_off - {l}  
        ,turn_on' = turn_on - {l}  
        ,last_state' = current_state  
        ,current_state' = current_state - {l}]"  
  
zoperation TurnOn =  
  over Dwarf  
  params leturn_on  
  pre "turn_off = {}"  
  update "[turn_off' = turn_off - {l}  
        ,turn_on' = turn_on - {l}  
        ,last_state' = current_state  
        ,current_state' = current_state  $\cup$  {l}]"
```

The pre-condition `pre "turn_off = {}"` is highlighted in yellow. The interface includes a menu bar (File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help), a toolbar, a file browser on the left, a console at the bottom, and a HyperSearch Results sidebar on the right. The status bar at the bottom shows the file path `57.22 (1515/3111)` and the session name `(isabelle,isabelle,UTF-8-Isabelle) | n m r o U G c. 1 t a`.

Checking the Specification

The screenshot shows the Isabelle2021-1/2_Machines - DwarfSignal.thy (modified) file in a text editor. The code defines a machine and a lemma:

```
is "???"  
  
zexpr DwarfReq  
is "NeverShowAll  $\wedge$  MaxOneLampChange  $\wedge$  ForbidStopToDrive  $\wedge$  DarkOnlyToStop  $\wedge$  DarkOnlyFromStop"  
  
zoperation ViolNeverShowAll =  
pre " $\neg$  NeverShowAll"  
  
zmachine DwarfSignalTest =  
init Init  
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll  
  
animate DwarfSignalTest  
  
lemma "Init establishes Dwarf_inv"  
by zpog_full
```

The right sidebar shows the HyperSearch Results Sidekick State Theories, with DwarfSignal selected. The bottom status bar indicates the proof state is 00% and the console output is empty.

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays the DwarfSignal.thy file with the following code:

```
is "???"  
  
zexpr DwarfReq  
is "NeverShowAll  $\wedge$  MaxOneLampChange  $\wedge$  ForbidStopToDrive  $\wedge$  DarkOnlyToStop  $\wedge$  DarkOnlyFromStop"  
  
zoperation ViolNeverShowAll =  
pre " $\neg$  NeverShowAll"  
  
zmachine DwarfSignalTest =  
init Init  
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll  
  
animate DwarfSignalTest  
  
lemma "Init establishes Dwarf_inv"  
by zpog_full
```

The right sidebar shows the HyperSearch Results, Sidekick, and State Theories. The State Theories panel lists 'DwarfSignal'.

The bottom console window shows the output of the simulation:

```
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10225630/simulate> ./Simulation  
Process  
/tmp/isabelle-simonfoster/process13243906329326067755/itree-simulate10168136/simulate/Simulation  
exited with code 834  
$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10225630/simulate>  
Starting ITree Simulation...  
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)  
SetNewProperState Drive;
```

The status bar at the bottom indicates 'Input/output complete' and '(isabelle,isabelle,UTF-8-Isabelle) | n m r o U G | 1 t n'.

Checking the Specification

The screenshot shows the Isabelle2021-1/2_Machines - DwarfSignal.thy (modified) file in a text editor. The code defines a system with various components and a lemma. The console output shows the execution of the simulation, including the start of the ITree simulation and the sequence of events.

```
is "???"  
zexpr DwarfReq  
is "NeverShowAll ^ MaxOneLampChange ^ ForbidStopToDrive ^ DarkOnlyToStop ^ DarkOnlyFromStop"  
zoperation ViolNeverShowAll =  
pre "¬ NeverShowAll"  
zmachine DwarfSignalTest =  
init Init  
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll  
animate DwarfSignalTest  
lemma "Init establishes Dwarf_inv"  
by zpog_full
```

System
/tmp/isabelle-simonfoster/process13243906329326067755/itree-simulate10168136/simulate/Simulation
exited with code 834
\$ISABELLE_TMP_PREFIX/process13243906329326067755/itree-simulate10225630/simulate>
Starting ITree Simulation...
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)
SetNewProperState Drive;
3
Internal Activity...
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L2;

Checking the Specification

The screenshot shows the Isabelle/ML editor interface. The main window displays the following code:

```
is "???"  
  
zexpr DwarfReq  
is "NeverShowAll  $\wedge$  MaxOneLampChange  $\wedge$  ForbidStopToDrive  $\wedge$  DarkOnlyToStop  $\wedge$  DarkOnlyFromStop"  
  
zoperation ViolNeverShowAll =  
pre " $\neg$  NeverShowAll"  
  
zmachine DwarfSignalTest =  
init Init  
operations SetNewProperState TurnOn TurnOff Shine ViolNeverShowAll  
  
animate DwarfSignalTest  
  
lemma "Init establishes Dwarf_inv"  
by zpog_full
```

The right sidebar shows the prover status: "Prover: ready" and "Continuous checking" is checked. Below this, there are buttons for "Purge", "HOL", "Scratch", and "DwarfSignal".

The bottom console window shows the following output:

```
System  
-----  
Starting ITree Simulation...  
Events: (1) Shine (Set [L1,L2]); (2) SetNewProperState Dark; (3) SetNewProperState Warning; (4)  
SetNewProperState Drive;  
3  
Internal Activity...  
Events: (1) Shine (Set [L1,L2]); (2) TurnOff L2;  
2  
Internal Activity...  
Events: (1) Shine (Set [L1]); (2) TurnOn L3;
```

The status bar at the bottom indicates "106.24 (2611/3111)", "Input/output complete", and "(Isabelle,Isabelle,UTF-8-Isabelle) | nmr o UGC 1 tn".

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.