A Tutorial of ZANS

— A Z Animation System

Xiaoping Jia
School of Computer Science, Telecommunication,
and Information Systems
DePaul University
Chicago, Illinois, U.S.A.
E-mail: jia@cs.depaul.edu
URL: http://venus.cs.depaul.edu/~xjia

Release 0.3
October 2002

1 What is ZANS?

ZANS is an animation tool for Z specifications. It is a research prototype that is still evolving.[1] This tutorial describes ZANS release 0.3.

The ultimate goals of ZANS are:

- Facilitate validation of Z specifications;
- Experiment design refinement and code synthesis based on Z specifications; and
- Assist learning of the Z specification language.

Currently, it supports the following features:

- type checking of Z specifications;
- expansion of schema expressions;
- evaluation of expressions and predicates;

[1] It is not thoroughly debugged and primitive in many aspects. Feedbacks and bug reports are appreciated.

- execution of operation schemas.

The input can be written in LaTeX with `zed` or `oz` packages, or in ZSL — an ASCII version of Z. The input forms and the type checking portion of ZANS are identical to those of ZTC version 2.03 [2]. Consult the *User's Guide* of ZTC version 2.03 for details of the input forms. A Z specification prepared for ZTC can be animated by ZANS with little or no modification. ZANS supports the Z syntax defined in the second edition of ZRM [3]. However, some deficiencies still remain in the current release of ZANS, most of which will be explained in more detail later in this document:

- The mathematical tools library is not user extensible.
- The animation is crude, not optimized.
- The animation mechanism is not very smart, yet.

These deficiencies will be addressed in the future versions of the tool.

Despite these deficiencies, many of my students and myself find the tool quite useful. Some of the student projects using ZANS will be made available.

2 Commands and Modes

To invoke ZANS, type `zans` on the command line. It will print out a brief message as follows and enter the *interpretation cycle.*

```
This is ZANS. Version 0.3.

... Initializing.
... Loading Z mathematical tools library: /usr/local/lib/zans/math1.zed

zans>
```

Now ZANS is waiting for a command. After you input a command, it will be interpreted and the results will be shown. Then the cycle will be repeated.

Note: Don't be alarmed if you see the following message when starting ZANS:

```
File not found: zans.cfg
No configuration file is found. Default input style is LaTeX.
```

If you are using the LaTeX input form you can proceed without any problem. If you use ZSL or LaTeX with `oz`, you should set up a configuration file (see section 5.2) before you start.

Not a lot of on-line help is available at this time. The `help` command will list all the commands supported by ZANS.

```
zans> help

analyze animate assign clear debug execute exit expand
expfile eval help list load para pragma pred
restart script show source state stop style trans
var verbose
```

To exit ZANS, use the `exit` command. This will bring you back to your operating system prompt.

After each session of ZANS, all the input and output of that session is saved in a log file named `zans03.log`. If you need to save the log, you have to rename the log file. Otherwise, it will be overwritten when you run ZANS next time.

2.1 ZANS command format

A ZANS command consists of three parts separated by one or more spaces:

command-name [*option*] [*arguments*]

The option and arguments are optional. The option part must begin with a hyphen (-). Here are some examples:

- `load classman`
  Command `load`, no option, argument `classman`.
- `execute -t Enrol` Command `execute`, option `-t`, argument `Enrol`.

Most of the commands are *single-line commands*. It means that when you hit the return key, it signals the end of the command and ZANS starts to interpret the command. There are also a number of *multi-line commands*. They are designed to allow lengthy arguments to the commands, such as expressions or paragraphs in a specification. For a multi-line command a single return key will not terminate the command, instead a continuation prompt `cont>` will appear. A multi-line command is terminated with two consecutive return's. Each command is clearly indicated whether it is a single or multi line command in Appendix A.

Here is a multi-line command that allows you to input a Z paragraph interactively.

```
zans> para
cont> Message ::= ok
cont>       | notok
cont>
... Type checking Free Type Definition: Message. "Stdin" Lines 1-2
```

2.2 ZANS operating modes

ZANS has two operating modes: the *initial mode* and the *animation mode*. The initial mode is the mode is the mode you are in when you start ZANS. The animation mode is the mode in which specifications can be animated. Most of the commands are available in both operating modes and will leave the operating mode unchanged. However, certain commands are only available in the animation mode. There are a pair of commands that will change the operating mode from one to the other:

- `animate`: from initial to animation.
- `clear`: from animation to initial.

The tow different modes are indicated by different command prompts:

- `zans>` indicates the initial mode.
- `anim>` indicates the animation mode.

3 Evaluating expressions and predicates

The `eval` command allow you to evaluate any Z expressions interactively. It is a multi-line command so that you can type in expressions that won't fit in one line. *Remember, you must terminate a multi-line command with two consecutive* return *'s.*

3.1 Simple expressions and predicates

The following command evaluates the set expression $\{x : 1 \mathinner{..} 13 \bullet 2 * x\}$.

```
zans> eval \{ x : 1 \upto 13 @ 2 * x \}
cont>

{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26}
```

Similar to the `eval` command, the `pred` command evaluates Z predicates. Here is a simple example to evaluate predicate $\{1, 2\} \subset (1 \mathinner{..} 7)$.

```
zans> pred \{ 1 , 2 \} \subset (1 \upto 7)
cont>

True
```

Here is a more complicated example: to check $\forall x : 1 \mathinner{..} 10 \bullet x \bmod 2 = 1$, i.e., whether every natural number between 1 and 10 is an odd number.

```
zans> pred \forall x : 1 \upto 10 @ x \mod 2 = 1
cont>

False
```

ZANS also allows you to introduce new variables and assign values to them interactively. To introduce new variables, you enter their declarations using the `para` command. The syntax is the same as the declaration part in a schema box or axiom box.

```
zans> para U, V, W : \power \nat
cont>
```

This declaration introduces three new names U, V and W, all of which are sets of natural numbers. The argument of the `para` command can be any Z paragraph.

You can assign values to the variable names using the `assign` command with the following syntax:

`assign` <*variable*> `:=` <*expression*>

Now we can assign values to U and V.

```
zans> assign U := \{ x : 1 \upto 5 @ x * x \}
cont>

{1, 4, 9, 16, 25}

zans> assign V := \{ x : 1 \upto 13 @ 2 * x \}
cont>

{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26}
```

Both `para` and `assign` are multi-line commands.

Now, you can check if the assignments are performed correctly by evaluating the variables as follows:

```
zans> eval U
cont>

{1, 4, 9, 16, 25}

zans> eval V
cont>

{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26}
```

Exactly what we expect! Now, let's assign W the intersection of U and V:

```
zans> zans> assign W := U \cap V
cont>

{4, 16}
```

All the types, operations, and relations defined in the ZRM *2nd edition* are supported by ZANS.

3.2 Function definitions

You can also define functions using λ-expressions.

```
zans> para AddOne : \nat \fun \nat
cont>

zans> assign  AddOne := ( \lambda x : \nat @ x + 1)
cont>

lambda x : N @ x + 1
```

Now you can apply the function to an expression.

```
zans> eval AddOne 2
cont>

3
```

Recursive functions also work fine.

```
zans> para Fact : \nat \fun \nat
cont>

zans> assign Fact := (\lambda n : \nat @ \zif n = 1 \zthen 1 \zelse  n * Fact(n-1) )
cont>

zans> eval Fact 4
cont>

24
```

3.3 Infinite sets

ZANS can handle infinite sets in several situations:

1. Membership in an infinite set.

```
zans> pred 2 \in \{ x : \num | x \mod 2 = 0 \}
cont>

True

zans> pred 3 \in \{ x : \num | x \mod 2 = 0 \}
cont>

False
```

2. Subset relation between a finite and an infinite set. (The left operand must be a finite set.)

```
zans> pred \{ 2, 4, 6 \} \subseteq \{ x : \num | x \mod 2 = 0 \}
cont>

True

zans> pred (2 \upto 5) \subseteq \{ x : \num | x \mod 2 = 0 \}
cont>

False
```

3. Intersection and difference of a finite and an infinite set. (The left operand must be a finite set.)

```
zans> eval \{ x : 1 \upto 100 | x \mod 3 = 0 \} \cap \{ x : \num | x \mod 2 = 0 \}
cont>

{6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96}

zans> eval \{ x : 1 \upto 100 | x \mod 3 = 0 \} \setminus
cont>\{ x : \num | x \mod 2 = 0 \}
cont>

{3, 9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99}

zans> eval (0 \upto 7 \cross 2 \upto 3) \cap \{ x, y : \num | x < y \}
cont>

{(0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}
```

ZANS represents set comprehensions in *closed forms*. It will be expand a set comprehension unless it's necessary. By default, the `eval` command does not exapnd set comprehensions.

```
zans> eval \{ x : 1 \upto 60  | x \mod 3 = 0 \}
cont>

{ x : 1 .. 60 | x mod 3 = 0 }
```

The `-e` option (*eager mode*) will force the expansion of set comprehensions.

```
zans> eval -e \{ x : 1 \upto 60 | x \mod 3 = 0 \}
cont>

{3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60}
```

It will cause an exception, if you force the expansion of an infinite set.

```
zans> eval \{ x : \nat | x \mod 3 = 0 \}
cont>

{ x : N | x mod 3 = 0 }

zans> eval -e \{ x : \nat | x \mod 3 = 0 \}
cont>

Execption: ZMT class error @ ZSetList::InsertElement().
Execeed maximun cardinality allowed.
{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57,
60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111,
114, 117, 120, 123, 126, 129, 132, 135, 138, 141, 144, 147, 150, 153, 156,
159, 162, 165, 168, 171, 174, 177, 180, 183, 186, 189, 192, 195, 198, 201,
204, 207, 210, 213, 216, 219, 222, 225, 228, 231, 234, 237, 240, 243, 246,
249, 252, 255, 258, 261, 264, 267, 270, 273, 276, 279, 282, 285, 288, 291,
294, 297}
```

To ensure termination of all evaluation and operations, ZANS enforce the following two limits:

- the maximum cardinality of a set (currently 100);
- the maximum iterations when expanding set comprehensions (currently 10,000).

4 Animation

4.1 Preparation

Animation is to execute the operation schemas in your specification to see if the specification accurately captures the requirements.

Before start animation, some minor modification to the specification may be necessary.

1. Indicate which schema is the state schema, and which is the initialization schema by adding the following tow lines into the specification

   %% `state-schema` *<state schema name>*
   %% `init-schema` *<initialization schema name>*

   These two lines may cause some harmless warning messages during type checking. Just ignore the warning message. If you do not have these tow lines in your specification, ZANS will attempt guess which one is the state schema and which one is the initialization schema. However, it is not very smart yet. With incorrect information on state and/or initialization schema, the animation may fail. Therefore, unless for very simple specifications, give ZANS a helping hand by specifying the state and initialization schemas explicitly to ensure that the animation can be carried out.

2. Optionally, you may also want to indicate which ones are user level operation schemas using the `operation` pragma.

   %% `operation Enquire`
   %% `operation Enrol`

```
%% operation Test
%% operation Leave
```

When `operation` pragmas are present, only these schemas indicated as operation schemas will be analyzed and can be animated. When `operation` pragma is absent, all the schemas will be analyzed and may be animated.

3. If you use global names, such as *size* in the Class Manager's Assistant example, you need to assign a specific value for the animation to be carried out. Append the following paragraph at the end of the Class Manager's Assistant specification should suffice:

   $$size = 6$$

4. Make implied predicate explicit. Originally, The schema *ClassInit* was defined as follows.

   $$ClassInit \mathrel{\widehat=} [Class' \mid enrolled' = \varnothing]$$

   ZANS will indicate that the schema is *non-explicit*, because $tested'$ is left unspecified. It is constrained by the invariant $\mathrm{dom}\, tested' \subseteq enrolled'$. We have to *infer* that the only value $tested'$ can take to satisfy the invariant is $\varnothing$. Therefore, the initial state is completely determined. However, ZANS is not smart enough to carried out this kind of inference.[2] Again a helping hand is needed here. Change the *ClassInit* to the following would suffice:

   $$ClassInit \mathrel{\widehat=} [Class' \mid enrolled' = \varnothing \land tested' = \varnothing]$$

Note: ZANS follows the convention of intialization schema in [1], *i.e.,* the initialization schema includes the dashed state schema.

4.2 Loading specifications

ZANS has extensive capabilities to manipulate schemas and schema expressions. Usually, you edit and save a specification in a file and type check it using ZTC. When the specification is correctly typed, it is ready for animation. In ZANS, you can load a specification with the `load` command.

[2]The current version of ZANS does not include a theorem prover. We are currently working on a theorem prover that will be integrated into ZANS in future releases.

```
zans> load classman.zed
Parsing main file: classman.zed
... Type checking Given Set. "classman.zed" Line 9
... Type checking Axiom Box. "classman.zed" Lines 13-15
... Type checking Free Type Definition: Response. "classman.zed" Lines 19-25
    Pragma 'state-schema', parameter: Class
... Type checking Schema Box: Class. "classman.zed" Lines 30-34
    Pragma 'init-schema', parameter: ClassInit
... Type checking Schema Box: ClassInit. "classman.zed" Lines 39-43
... Type checking Schema Box: Enrolok. "classman.zed" Lines 52-61
... Type checking Schema Box: Testok. "classman.zed" Lines 64-73
... Type checking Schema Box: Leaveok. "classman.zed" Lines 76-86
    Pragma 'operation', parameter: Enquire
... Type checking Schema Box: Enquire. "classman.zed" Lines 91-98
... Type checking Schema Box: AlreadyEnrolled. "classman.zed" Lines 102-108
... Type checking Schema Box: NoRoom. "classman.zed" Lines 111-117
... Type checking Schema Box: AlreadyTested. "classman.zed" Lines 121-127
... Type checking Schema Box: NotEnrolled. "classman.zed" Lines 130-136
    Pragma 'operation', parameter: Enrol
    Pragma 'operation', parameter: Test
    Pragma 'operation', parameter: Leave
... Type checking Schema Definition: Enrol. "classman.zed" Line 144
... Type checking Schema Definition: Test. "classman.zed" Line 145
... Type checking Schema Definition: Leave. "classman.zed" Line 146
End of main file: classman.zed
```

This loads the specification of a *Class Manager's Assistant* in [1]. I will use this specification as an example in the remainder of this tutorial. The specification is type checked when it is loaded. The type checking performed here is identical to ZTC version 2.03.

Before you proceed, you can list all the schema names in the specification with the `list` command:

```
zans> list

Class
ClassInit
Enrolok
Testok
Leaveok
Enquire
AlreadyEnrolled
NoRoom
AlreadyTested
NotEnrolled
Enrol
Test
Leave
```

They are listed in the order they appear in the specification. Now you can expand the schemas using the `expand` command which takes a schema expression as its parameter.

```
zans> expand ClassInit
cont>

--- ClassInit ------------------------------------------
|  enrolled', tested' : P Student
|-------------------------------
|  enrolled' = {};
|  tested' = {};
|  # enrolled' <= size;
|  tested' subseteq enrolled'
--------------------------------------------------------
```

4.3 Analyzing specifications

As you see, ZANS can only animate a subset of Z specifications, however I will argue that it is a very useful subset, and the kind of modifications needed do not sacrifice the virtues of Z specifications.

The mechanism behind animation is called *explicitness analysis*. ZANS can only animate those operation schemas that are *explicit*. Informally, an operation schema is explicit if all the output variables and the post-state are defined by the input variables and the pre-state. For details of the animation mechanism, see [4].

When ZANS indicates that a schema is non-explicit, there are several possible causes:

- A missing post-condition in the schema. In this case, ZANS actually reveals a problem in the specification.
- An implicit post-condition, like the one in *ClassInit*. Make the implied post-condition explicit as illustrated above, and try again.[3]
- An intermediate schema not intended to be executed directly. Ignore the non-explicit message.
- Intentionally loose specification. ZANS can not animate most of the loose specifications.

Schemas that are non-explicit can still be animated, but the output will contain undefined values.

Now, to start the animation, use the `animate` command. This will cause ZANS to enter the animation mode:

```
zans> animate
... Initialization.
```

The initialization of animation may take a while depending on the load of the system and the size of your specification. A lot of things are happening during initialization. You will see a lot of messages as it progresses. Don't worry, if you didn't catch all the messages. All the screen output is saved in the log file `zans03.log`.

The messages show the progress of the initialization process. The initialization process begins with looking all the schema classification pragmas:

[3]Implicit post-conditions arise in many situations. For example, x' can be constrained by x as follows:

- function inversion: $x = f(x')$.
- equation: $F(x, x') = 0$,

It is not always possible to derive f^{-1} from f or to solve a general equation.

```
- Search for schema classification pragmas.
  pragma 'state-schema', parameter: Class
  pragma 'init-schema', parameter: ClassInit
  pragma 'operation', parameter: Enquire
  pragma 'operation', parameter: Enrol
  pragma 'operation', parameter: Test
  pragma 'operation', parameter: Leave
```

Now, ZANS analyzes the state and the initialization schemas. If `state-schema` or `init-schema` is not present, ZANS will attempt to *auto-set* the state and initialization schemas. Auto-set can be fooled sometimes. Using explicit `state-schema` and `init-schema` is the safest bet.

```
- Analyze state schemas.
    state schema: Class -- ok.
  State schema analysis done.
- Analyze initialization schemas.
    init schema: ClassInit -- ok.
  Initialization schema analysis done.
```

Next, ZANS analyzes the axiomatic definitions to see if all the global names are properly defined.

```
- Analyze axiomatic definitions -- ok.
```

If *size* were not defined in our example, it would be non-explicit.

Next, ZANS will perform explicitness analysis on each operation schema specified by the `operation` pragma.

```
    analyze operation schema: ClassInit -- ok.
    analyze operation schema: Enquire -- ok.
    analyze operation schema: Enrol -- ok.
    analyze operation schema: Test -- ok.
    analyze operation schema: Leave -- ok.
```

In this example, all the operation schemas are explicit.

Before animating the specifition the global environment must be set. All global names must be initialized.

```
... Initializing equivalence definitions.
... Initializing global names.
### Try branch #1
### Branch #1 succeed.
size : 6
```

Next, ZANS executes the initialzation schema.

```
    Initialization schema ClassInit
... Execute schema: ClassInit
### Try branch #1
### Branch #1 succeed.
Schema: ClassInit
enrolled':  {}
tested':  {}

anim>
```

Now, ZANS is ready for your commands. Note that the prompt is changed to `anim>` as we are now in the animation mode.

4.4 Get set, Go!

You can execute any of the explicit operation schemas using the `execute` command. You will be prompted for input arguments when needed.

First, we enroll *Jack* in the class.

```
anim> execute Enrol
... Execute schema: Enrol
Enter input arguments:
s? -> Jack
### Try branch #1
### Branch #1 succeed.
Schema: Enrol
enrolled:  {}
tested:  {}
enrolled':  {Jack}
tested':  {}
s?:  Jack
r!:  success
```

We can have a look at the current state using the `show -v` command.

```
anim> show -v Class
Schema: Class
enrolled:  {Jack}
tested:  {}
```

By default, the execution of an operation schema will update the values of the components in the state schema. If you only want to see the execution result of an operation schema but do not want to update the state, you can execute the operation schema in *trial mode*, or *non-committal* mode, with the `-t` switch.

```
anim> execute -t Enrol
... Execute schema: Enrol
Enter input arguments:
s? -> Jill
### Try branch #1
### Branch #1 succeed.
Schema: Enrol
enrolled:  {Jack}
tested:  {}
enrolled':  {Jack, Jill}
tested':  {}
s?:  Jill
r!:  success
```

To verify that the state is not changed:

```
anim> show -v Class
Schema: Class
enrolled:  {Jack}
tested:  {}
```

We can continue to enroll *Jill*, *Mark*, *Jennifer*, *Susan*, and *Johnson*. Now the state is the following:

```
anim> show -v Class
Schema: Class
enrolled:  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested:  {}
```

Now, *Jill* has passed the test.

```
anim> execute Test
... Execute schema: Test
Enter input arguments:
s? -> Jill
### Try branch #1
### Branch #1 succeed.
Schema: Test
enrolled:  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested:  {}
enrolled':  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested':  {Jill}
s?:  Jill
r!:  success
```

Now, we can enquire the status of *Jack*.

```
anim> execute Enquire
... Execute schema: Enquire
Enter input arguments:
s? -> Jack
### Try branch #1
### Branch #1 fail.
### Try branch #2
### Branch #2 succeed.
Schema: Enquire
enrolled:  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested:  {Jill}
enrolled':  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested':  {Jill}
s?:  Jack
r!:  alreadyenrolled
```

Another way to execute an operation schema is to use the *try-all mode* with the `-a` switch. It will try all the branches to see if the operation is loose.

Now, we try to enrol *Susan* again, using try-all mode.

```
anim> execute -a Enrol
... Execute schema: Enrol
Enter input arguments:
s? -> Susan
### Try branch #1
### Branch #1 fail.
### Try branch #2
### Branch #2 succeed.
Schema: Enrol
enrolled:  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested:  {Jill}
enrolled':  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested':  {Jill}
s?:  Susan
r!:  noroom
### Try branch #3
### Branch #3 succeed.
Schema: Enrol
enrolled:  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested:  {Jill}
enrolled':  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested':  {Jill}
s?:  Susan
r!:  alreadyenrolled
>>> Operation Enrol is loose. 2 branches succeeded.
```

The try-all mode is always non-committal. Now, *Susan* leaves the class.

```
anim> execute Leave
... Execute schema: Leave
Enter input arguments:
s? -> Susan
### Try branch #1
### Branch #1 fail.
### Try branch #2
### Branch #2 succeed.
Schema: Leave
enrolled:  {Jack, Jill, Mark, Jennifer, Susan, Johnson}
tested:  {Jill}
enrolled':  {Jack, Jill, Mark, Jennifer, Johnson}
tested':  {Jill}
s?:  Susan
r!:  cert
```

Oops. *Susan* is not supposed to get a certificate. Double check the specification, you will find a mistake in the *Leave* schema. This is just one kind of problems in Z specifications that can be discovered with the help of ZANS.

5 Extra Points

5.1 Compiled specification

If you are puzzled by the "non-explicit" complaints, you may want to have a look at the "complied" specification. It offers some clues as to why an operation schema is not explicit. Use

`analyze` *filename*

to generate the compiled specification and save it in a file named *filename*. The default extension is `ope`. The compiled specification is represented in the *extended guarded commands* described in [4].

For an operation schema that is not explicit, you should find messages like:

`Unable to find definition for:` *variable*

This indicates that either the definition concerning *variable* is missing, or *variable* is implicitly constrained not explicitly defined.

Sorry, if that's not much help at all. It's only version 0.3.

5.2 Configuration file

If you use ZSL or LaTeX with `oz` mnemonics, you need to set up a *configuration file*. The configuration file is named `zans.cfg`. It is simply an empty specification in the style you choose.

For ZSL, it is

```
specification
end specification
```

Remember, there must be a leading TAB on each line.

For LaTeX `oz-zed` compatible mode, it is

```
%% oz
\begin{spec}
\end{spec}
```

For LaTeX `oz` native mode, it is

```
%% oz-native
\begin{spec}
\end{spec}
```

If the configuration file is not found, the default input form is LaTeX with only the mnemonic names defined in the `zed.sty` package.

5.3 Creating and running scripts

During specification validation, you may need to run a sequence of commands repeatedly. ZANS allows you to save the commands in a *script file*. A script file is simply an ASCII file that contains ZANS commands. Each line contains a single command. A multi-line command must be followed by a blank line. Script files can be created using a text editor, or the `script` command in ZANS, which records the commands you type during an interactive session.

To record a script file, first you specify the script file name as follows:

`script` *spcriptfile*

Now the recording starts. All the following commands you type will be saved in the script file name *spcriptfile*. The default extension of script files is `spt`.

You can turn the recording on and off with the following commands:

`script -off` to turn off

`script -on` to turn on

To run a script file, you use the `source` command:

`source` *spcriptfile*

Alternatively, you can directly invoke ZANS with a script file from the command line using the `-f` option:

`zans -f` *spcriptfile*

5.4 Verbosity control

If you want to see how does ZANS animate your specifications, you can set a higher verbosity level, so that ZANS will reveal a lot more about the animation process.

You can adjust the verbosity level using the `verbose` command:

`verbose` *digit*

The verbosity level ranges from 0 to 9, with 9 being the most verbose and 0 the least. The initial verbosity level is set to 5. You can set your own initial verbosity level at the beginning of the configuration file with the `%% verbose` pragma.

References

[1] J.B. Wordsworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley, 1992.

[2] Xiaoping Jia. *ZTC: A Type Checker for Z Notation, User's Guide, Version 2.02*, June 1995. Available via anonymous ftp at `ise.cs.depaul.edu`.

[3] J.M. Spivey. *The Z Notation, A Reference Manual.* Prentice Hall International, second edition, 1992.

[4] Xiaoping Jia. An approach to animating Z specifications. In *Proc. 19th Annual Int'l Computer Software and Applications Conf.*, Dallas, Texas, USA, August 1995.

A Summary of ZANS Commands

Notation.
Each command is annotated using the following notation:

- S: single-line command; M: multi-line command.
- I: available in the initial mode; A: available in the animation mode.
- →I: transition to the initial mode; →A: transition to the animation mode.

Examples:

- <M,IA>: multi-line command, available in both modes, no mode change.
- <S,I→A>: single-line command, available only in the initial mode and changes to the animation mode after its execution.

Summary of ZANS Commands

`analyze` *filename* <S,IA>
Analyze the entire specification. The operations generated are saved in the file named *filename*. Default extension is `.ope`.

`animate` <S,I→A>
Start animation.

`assign` *variable* `:=` *expression* <M,IA>
Assign the value of the *expression* to the *variable*.

`clear` <S,IA→I>
Clear the current specification.

`eval` *expression* <M,IA>
Evaluate the *expression*.

Options:
`-e`: eager evaluation

`execute` [`-at`] *schemaname* <S,A>
Execute the operation schema named *schemaname*.

Options:
`-a`: try all branches
`-t`: non-committal execution

`exit` <S,IA>
Exit ZANS.

`expand` [`-dn`] *schemaexp* <M,IA>
Expand the schema expression.

Options:
`-d`: convert to disjunctive normal form
`-n`: normalized

`expfile` [`-dn`] *filename* <S,IA>
Expand the entire specification and save the results in the file named *filename*.

Options:
`-d`: convert to disjunctive normal form
`-n`: normalized

`help` <S,IA>
List all the commands of ZANS.

`list` <S,IA>
List all the schema names in the currently loaded specification.

`load` *infile* <S,IA>
Load and type check the specification file named *infile*.

`para` *paragraph* <M,IA>
Enter and type check a paragraph.

`pragma` *pragmaname* [*args* ...] <S,IA>
Set pragmas.

`pred` *predicate* <M,IA>
Evaluate the *predicate*.

`script` *scriptfilename* <S,IA>
Set the name of the script file to *scriptfilename*.

`script` [`-on`|`-off`] <S,IA>
Turn on or off the script file.

`show` [`-ov`] *schemaname* <S,IA>
Show the schema named *schemaname* in its original, unexpanded form.

Options:
`-o`: show the operation generated from the schema;
`-v`: show the current value of the schema components.

`source` *scriptfilename* <S,IA>
Run the script file named *scriptfilename*.

`stop` <S,A>
Stop animation.

`style` [`-tlb`] <S,IA>
Set the output style

Options:
`-t`: LaTeX style
`-l`: ZSL text style
`-b`: ZSL box style (default)

`verbose` *digit* <S,IA>
Set verbosity level: 0-9. 0 the least verbose, 9 the most verbose.
Initial value: 5.