Using the ISM Framework

Sebastian Nanz and David von Oheimb

15th September 2003

Abstract

This manual specifies the various ISM representations and explains the use of the ISM framework within AutoFocus and Isabelle/HOL. This includes: installing the converter tool for AutoFocus, extending Isabelle with ISM sections, modeling ISMs in AutoFocus, translating ISMs from AutoFocus to Isabelle/HOL, defining ISMs directly as theory sections, and accessing the resulting definitions.

This text is based on extracts from [Nan02, Part II].

1 Installation

1.1 Installing the Converter Tool

The converter comes in two flavors: the *plug-in version* is integrated in AutoFocus, which has to be installed beforehand. It is use simply by selecting the right menu entry in AutoFocus. The plug-in version is recommended for most occasions.

The *standalone version* is a command line application. An installation of AutoFocus is not strictly necessary to use it, however two additional `.jar`-files are needed that are part of the AutoFocus distribution. If one wants to create or change models, an installation of AutoFocus will be necessary. The standalone version is recommended if only the translation feature is needed and no own models will be developed in AutoFocus.

Whenever necessary in the following sections, procedures for the plug-in and the standalone version will be described separately.

1.1.1 Plug-in

- If AutoFocus is not installed on the target system yet, download a copy of the latest distribution from the AutoFocus web site[1] and install it according to the installation instructions coming with the distribution. Let *af_home* denote the AutoFocus home directory created during the installation process (typically `$HOME/AutoFocus`).

- Add the following line to the file *af_home*/`Extensions.ecf`:

  `Isabelle Theory;` *isa_models*`; quest.isabelle.exporter.IsaExporterWrapper`

[1] http://autofocus.in.tum.de/

where *isa_models* is the directory where the temporarily exported `.qml`-files will be saved. It is recommended to set this to an absolute path (e.g. `/home/nanz/tmp`) rather than to a relative one (`tmp`) – if choosing the relative version, a new subdirectory (e.g. `tmp`) will be created for every export in the directory from with AutoFocus has been *started*.

- Unzip the archive that contains the converter (`Isabelle.zip`) into the directory *af_home*`/Applications/Quest/`. This will create `Isabelle.jar` and a subdirectory `isa-settings` populated with some configuration files.
- One can now start AutoFocus as described in the AutoFocus documentation. The AutoFocus plug-in mechanism will have created a new menu entry:

  `Project->Export Project->Isabelle Theory`

1.1.2 Standalone Application

- Unzip `Isabelle.zip` into a directory. This will create `Isabelle.jar` and a subdirectory `isa-settings` populated with some configuration files.
- Make sure that both the files `sim.jar` and `log4j.jar` exist; these are part of the AutoFocus distribution, contained in the directory *af_home*`/Applications/Quest/`, where *af_home* denotes the AutoFocus home directory created during the installation process.
- Extend the `CLASSPATH` with the `.jar`-files `Isabelle.jar`, `sim.jar`, and `log4j.jar`, i.e. add *full_path_to_jar_file*/*jar_file* to the path for each of the three `.jar`-files (on UNIX systems these paths have to be separated by colons `:`).
- One can now start the exporter by typing the following command in a shell:

  `java quest.isabelle.exporter.IsaExporterWrapper`

Of course it is possible to use the converter tool both as a plug-in and as a standalone version with one installation. Simply follow the installation instructions for the plug-in and in addition extend the `CLASSPATH` as described in the standalone installation version.

1.2 Preparing Isabelle

If Isabelle is not installed on the target system yet, download a copy of the latest distribution from Isabelle's web site[2] and install it according to the installation instructions coming with the distribution.

In order to use the **ism** section (see section 4) the following steps have to be performed:

- Make sure that the following files exist on the system: `ISM_package.ML` (ism theory extension), `ISM_package.thy` (loads the package), and the theories that define the semantics for the ISMs, i.e. at least `Basis.thy` and `ISM.thy`. Place these files in one directory together with the theory file to be developed, e.g. an converted model produced by the converter tool.

[2] `http://isabelle.in.tum.de/`

- If using the Proof General interface the following additional steps have to be performed:
  - Make a backup copy of `isar-keywords.el`, to be found in `$ISABELLE_HOME/etc/` (please refer to the Isabelle System Manual [BW02]).
  - Start Isabelle with the Proof General interface. Open and load ("use") the theory `ISM_package.thy`. The package `ISM_package.ML` will be read.
  - Execute the following command in Proof General:
    `ML {* ProofGeneral.write_keywords "" *}`.
    This will create another file `isar-keywords.el` in the current working directory with the **ism** section keywords added.
  - Copy this file to the original location of `isar-keywords.el`.
  - Restart Proof General. The section keywords will be recognized now.

2 Modeling with AutoFocus

A tutorial on modeling with AutoFocus can be found on the web site cited before. Please refer to this document for questions on AutoFocus usage.

All models created in AutoFocus can be exported "as is" using the converter tool. Yet in order to obtain a runnable Isabelle theory it might be necessary to adapt the model using the remarks and hints given below; warnings and errors during the export will show if this is necessary or not. The following remarks should be definitely beared in mind, they are necessary for a successful export.

- It should be ensured that the project selected for export is a Quest project and not a Java project. Java projects are not supported.
- The types of messages (i.e. the types of the ports and channels) must be constructor-based datatypes defined in an DTD of the project. No predefined types such as `Int` and `Bool` are allowed; if they occur the error "`Port ... has type ... without TypeDef (predefined types cannot be used for translation). Messages type possibly incomplete.`" is thrown.

  This is necessary because in the translation ISM have to be parametrized with exactly one message type in order to communicate.

  Note: If multiple constructor-based datatypes exist, the union of the different constructors will build a new message datatype.

  Hint: If starting from scratch, it is highly recommended to define only one datatype and use it on all ports and channels.

  Hint: If a model contains ports and channels with predefined types, the following will be sufficient to ensure a successful export: add a constructor for each predefined type (e.g. `IntVal(Int)`) to a new message type and replace the old type on all affected ports and channels with the new one. Replace all messages m of the old type with the constructor applied to the messages (e.g. `IntVal(`m`)`).

- In STDs exactly one initial control state is allowed, otherwise the error "`Found more than one initial control state.`" is thrown.

  Hint: If there is more than one initial control state, it is sufficient to introduce a new super-initial state with non-deterministic transitions to the old initial states.

If starting from scratch, the following hints will be very useful but they are not necessary for a successful translation.

- Spaces and special characters in names should be avoided. In the translation such characters will be replaced with an underscore and the warning "`Changed identifier ... to ....`" will be thrown.
- The usage of unique port names is recommended. ISM ports merge the notion of AutoFocus ports and channels; the name of the ISM port will be determined from the last AutoFocus port (that has no output ports) in a directed chain of channels.

  Note: If a port name occurs twice, it will be made unique by appending a number to its name.
- If an AutoFocus component has n channels attached to the same output port, in the translation to an ISM the corresponding n input ports will appear. Currently, the output patterns in the transition will be replicated accordingly rather than making use of the multicast pattern available for ISMs.
- The usage of unique control state names is recommended. The names of the control states will be used as constructors for the Isabelle datatype of control states of the ISM. Clashes occur if similar constructor names are used for different datatypes in the same theory. Setting the switch `DATATYPE_LONG_NAMES` in the `Settings` file (see section 3.1) to `true` will avoid these clashes by usage of prefixed constructor names, e.g. `A_control.Main` instead of just `Main` if `Main` is a constructor of the datatype `A_control`. But since this is done in all occurrences of constructors in terms one might want to avoid this completely.
- When using hierarchical components in SSDs, note that if a component has both a sub-structure and an automaton, the automaton will be ignored. Most Quest tools follow this policy.
- Note that hierarchy in STDs is not supported.
- Note that the central concept in the translation is the component. For example if an STD is not assigned to some component, it will not be exported.
- There is no way to export EETs.

3 Translation to an Isabelle Theory

3.1 Configuration Files

The converter tool comes with a set of configuration files located in the subdirectory `isa-settings` of the installation directory. These files can change the appearance of the generated theories.

Note: In order to have effect on the generation of the next theory exported, the plug-in mechanism might require AutoFocus to be shut down and restarted again. The Java Virtual Machine will not recognize updates on files on disk if they are still in memory.

Note: The escape character in all files is the backslash. Entries are split on whitespace, to avoid this, entries can be grouped by double quotes.

3.1.1 Settings

The `Settings` file provides a set of configuration variables to change the environment of the exporter tool, i.e. location of important files and directories, to provide switches for theory generation, and to change user-defined theory constants.

The names and location of the following files and directories can be changed (paths to configuration files are relative to the `isa-settings` directory):

`PRELUDE_DTD_FILE_NAME` Name of the PreludeDTD file (see below). Default: `generated/PreludeDTD`

`PRELUDE_DTD_DATA_FILE_NAME` Name of the additions file for the PreludeDTD (see below). Default: `PreludeDTDAdditions`

`OUTER_SYNTAX_FILE_NAME` Name of the Isabelle keywords file (see below). Default: `OuterSyntax`

`SYMBOL_FILE_NAME` Name of the symbols file (see below). Default: `Symbols`

`LOG_CONFIG_FILE_NAME` Name of the log4j configuration file: Default: `logging.cfg`

`DEFAULT_SAVE_PATH` Absolute path of the directory where generated theory files should be saved to. Typically a directory in `$HOME`.

The following switches (values `true` and `false`) affect the appearance of generated theories:

`READ_PRELUDE` Whether the PreludeDTD file (see below) should be read in or not. Default: `true`

`GENERATE_FLAT_ISM_COMPOSITIONS` Whether flat (`true`) or hierarchic (`false`) compositions should be generated. Default: `true`

`CHECK_VALID_IDENTIFIERS` Whether identifiers should be checked and corrected if clashes with Isabelle keywords occur or special characters are contained. Default: `true`

`X_SYMBOLS_ON` Whether X-Symbols should be generated (`true`) or rather ASCII symbols (`false`). Default: `true`

`DATATYPE_LONG_NAMES` Whether constructors should occur with prefixed name in terms or not, e.g. `A_control.Main` instead of just `Main` if `Main` is a constructor of the datatype `A_control`, see also section 2. Default: `false`

`GENERATE_DISCRIMINATORS` Whether to generate discriminator constants. AutoFocus generates constants of the form `is_`*constr* for each constructor *constr* of user-defined datatypes; these can be exported as Isabelle/HOL **consts** declaration. Note: Definitions for these constants are not generated within AutoFocus and therefore also not available in the exported theory. Default: `false`

`GENERATE_SELECTORS` Whether to generate selector constants *constr_i*`Sel`*i*. Analogous to `GENERATE_DISCRIMINATORS`. Default: `false`

Some variables affect certain names used in the theory.

`ISM_BASIS_THEORY` Name of the theory on which the generated theory should be based on. This is usually `ISM_package` or a theory that requires `ISM_package` itself. Default: `ISM_package`

`ISM_PT_TYPE` Name of the global port type *pt_type*. Default: `port`

`ISM_MSG_TYPE` Name of the global message type *msg_type*. Default: `message`

`ISM_ST_TYPE` Name of the global state type *st_type*. Default: `global_state`

`DATA_VAR_NAME` Name of the data state variable *data_st_var*. Default: `s`

3.1.2 PreludeDTD

This is a generated file and is updated every time when starting an export, provided that the switch `READ_PRELUDE` is set to `true`. Then the DTD module defined in the file will be added to the modules in the exported project with the result that definitions and declarations are available when terms are checked. Thus it is possible to avoid redefinition of functions already defined in Isabelle while constants will be still recognized as constants. Note: If type checking transitions, declaration of all occurring functions in AutoFocus will not be avoidable, since the addition of the Prelude DTD module is at export time.

The entries generated in the `PreludeDTD` come from two sources:

- Symbol table entries (see below) marked as `term` (namely a function or constant) and `undef` (not predefined in AutoFocus): For example the symbol table entry

  ```
  elem  term  2  :  \<in>  infixl  50  undef
  ```

  will result in the line

  ```
  fun elem(a0, a1) = 0;
  ```

  Only the arity of the operator is used to generate the dummy definition since no type check occurs at this moment.

- Quest language code written in the `PreludeDTDAdditions` file (see below) is placed in the prelude without modifications.

3.1.3 Symbols

The entries in this file affect the translation of symbols by changing of the associativity and precedence of operators and type constructors and the actual symbol representation. This is necessary since symbols in AutoFocus differ in many points:

Example 3.1 (Precedence and Associativity)
Priorities in AutoFocus range from 1 (weak) to 4 (strong), associativity is defined for infix operators and can be to the left or the right [BLS00]. Priorities in Isabelle range from 0 (weak) to 1000 (strong), and for arbitrary mixfix constants, associativities can be defined [Pau02] (we consider only associativity to the right and the left).

The conjunction is now written in AutoFocus as `&&` and has precedence 2 and associates to the left. In Isabelle the conjunction is written as ASCII symbol `&` and X-Symbol ∧ (`\<and>`) associating to the right. Thus the entry in the symbol table will look like this:

```
&&  term  2  &  \<and>  infixr  35  predef
```
□

Since the term tree is already constructed by the Quest importer, only Isabelle relevant data is necessary in the table.

Entries in the symbol table are single lines with the following 8 entries from left to right, separated by whitespace.

- The symbol used in AutoFocus.
- One of the keywords `term` or `type`, depending on the symbol being an operator or a type, respectively.
- Arity of the symbol (equal in AutoFocus and Isabelle).
- The corresponding ASCII Isabelle symbol.
- The corresponding X-Symbol or a repetition of the ASCII Isabelle symbol.
- One of the following keywords related to the Isabelle symbol:

  `nofix` Not an infix symbol.
  `infixl` Infix symbol. Association to the left.
  `infixr` Infix symbol. Association to the right.
  `infix` Infix symbol. No associativity. Example: `<`
  `infixswap` Infix symbol which arguments have to be swapped. Example: Translation of `>` to `<`
  `prefix` Prefix symbol.

- The Isabelle operator precedence, ranging from 0 (weak) to 1000 (strong).
- One of the keywords `predef` or `undef`, depending on the symbol being predefined in AutoFocus or not, respectively.

Example 3.2 (Symbol Table Entries)
Further examples how symbol table entries can look like:

```
Bool  type  0  bool  bool        nofix   1000  predef
&&    term  2  &     \<and>      infixr    35  predef
elem  term  2  :     \<in>       infixl    50  undef
map   type  2  ~=>   \<leadsto>  infixr     0  undef
```
□

3.1.4 PreludeDTDAdditions

The file contains code that is placed in the `PreludeDTD`, see above.

3.1.5 OuterSyntax

Each line of this file contains a single Isabelle keyword like **datatype** or **ism**. New keywords can be appended at the end of the file. For a complete update, the file `isar-keywords.el`, be found in `$ISABELLE_HOME/etc/`, has to be referenced.

3.2 Structure of the Generated Theory

The converter tool tries to export all SSDs of the project and all sub-components recursively. Each SSD "tree" will be exported to an own theory file. STDs will be exported if assigned to a SSD. All DTDs will be exported.

The structure of a generated theory looks like shown in figure 1.

theory . . . = . . .

Type declarations and definitions exported from DTDs

typedecl . . .
datatype . . .

Type definitions and abbreviations for the global types

datatype *message* . . .
types *state* . . .
datatype *port* . . .

Constants and constant definitions

consts . . .
constdefs . . .
consts . . . **axioms** . . .

ISMs and parallel compositions

ism . . .

end

Figure 1: Structure of a Generated Theory

3.3 Export with AutoFocus Plug-In

To export a project using the plug-in version of the tool, select the project from the project tree and choose `Export Project -> Isabelle Theory` from the `Project` menu as shown in figure 2.

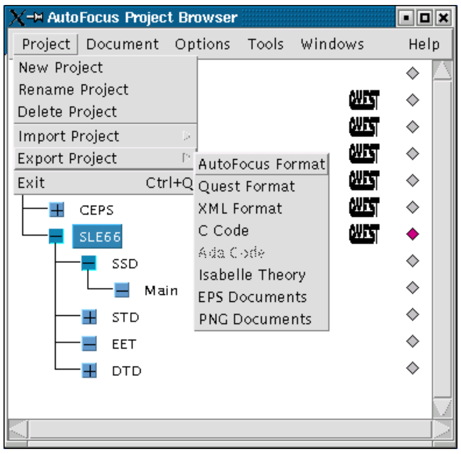

Figure 2: Snapshot of the AutoFocus Export Menu

3.4 Export with Standalone Application

To start the standalone version of the tool, type

```
java quest.isabelle.exporter.IsaExporterWrapper
```

with the necessary arguments in a shell. If no arguments are given, the following usage message will appear:

```
Usage: java quest.isabelle.exporter.IsaExporterWrapper [OPTIONS] FILE
Options:
  -h,      --help              print this help message
  -p BOOL, --prelude BOOL      use prelude DTD file: true/false
  -a,      --ascii             use ascii symbols only (no x-symbols)
  -c BOOL, --flat-comp BOOL    create flat compositions of ISMs
                               (else hierarchic): true/false
  -v BOOL, --validation BOOL   check if identifiers are valid: true/false
  -b PATH, --base-path PATH    set PATH to "isa-settings" directory
  -l BOOL, --long-names BOOL   expand datatype names: true/false
Arguments:
  FILE                         file in Quest format (.qml)
```

Note: The command-line flags overwrite the flags described in section 3.1. This is especially useful for fast testing of the appropriate settings.

4 ism Theory Section

4.1 Description

In order to facilitate the usage of ISMs, a new theory section has been defined. Theory sections in Isabelle start with a keyword (for example **consts**, **inductive**, or in this case **ism**) and can extend the current theory by new definitions and theorems. In this case, for each ISM a record with all necessary fields is created. This includes the set of possible translations, which is defined inductively.

This section describes the syntax and semantics of the individual parts of an ISM section. For further motivation and more details of the ISM semantics see [Ohe02, ?].

A description of the syntactic structure of the section in an BNF-like style can be found below. For the sake of readability we have simplified the specification; it is correct with the following additional remarks:

- At least one of the subsections **control** or **data** has to be present.
- The list elements within the subsections **pre**, **in**, **out**, and **post** are separated by commas.
- Control state transitions can be specified only if the subsection **control** is present.
- Postconditions with the subsection **post** can be specified only if the subsection **data** is present.
- Instead of `->` one may also use the X-Symbol $\rightarrow$.

```
ism name ((param_name :: param_type))* =
  ports pn_type
    inputs   I_pns
    outputs  O_pns
  messages msg_type
 [commands cmd_type [default cmd_expr']]
   states    [state_type]
   [control cs_type [init cs_expr0]]
   [data     ds_type [init ds_expr0] [name ds_name]]
 [transitions
   (tr_name [attrs]: [cs_expr -> cs_expr']
   [pre  (bool_expr)+]
   [in    ([multi] I_pn  I_msgs)+]
   [out  ([multi] O_pn O_msgs)+]
   [cmd    cmd_expr]
   [post ((lvar_name := expr)+ | ds_expr')]
   )+]
```

The meaning of the individual parts is as follows.

- The ISM definition will be referred to by *name*. It may have any number of parameters, each declared by *param_name* and its corresponding type *param_type*. The parameters may be used throughout the definition body.

- The type expression *pn_type* gives the Isabelle/HOL type of the port names, while *I_pns* and *O_pns* denote the set of input and output port names, respectively. If ports can be changed dynamically, like with dynamic ISMs, the sets given here specify the initial or maximal interface.

- The type expression *msg_type* gives the type of the messages, which is typically an algebraic datatype with a constructor for each kind of message.

- The optional *cmd_type* specifies the type of ISM commands. It must be given if commands are used in the transitions. The optional default command *cmd_expr'*, which typically is the empty list of commands, can be used to shorten the specification of transitions that do not actually issue commands.

- The optional *state_type* should be given if the current ISM forms part of a parallel composition and the state types of the ISMs involved differ. In this case, *state_type* should be a free algebraic datatype with a constructor for each state type of the ISMs involved.
  The type expressions *cs_type* and *ds_type* give the types of the control and data state, respectively, while the optional terms *cs_expr0* and *ds_expr0* specify their initial values — if not given, they default to some arbitrary value. Either (i.e., not both) the control state or the data state may be absent.
  The optional logical variable name *ds_name*, which defaults to `s`, may be used to refer to the whole data state within transition rules.

Transitions are given via named rules where *attrs* is an optional list of attributes, e.g. [**intro**]. The rule can be accessed after defining the ISM via *name*.**transs.***tr_name*. The control states (if any) before and after the transition are specified by the expressions[3] *cs_expr* and *cs_expr'*.

Expressions within a rule may refer to the logical data state variable mentioned above. In particular, assuming that `s` is the name of the data state variable, then the value of any local variable `lvar` of the ISM may be referred to by `lvar s`. The scope of free variables appearing in a rule is the whole rule, i.e. free variables are implicitly universally quantified (immediately) outside each rule.
All the following parts of a transition rule are optional:

- The **pre** part contains guard expressions *bool_expr*, i.e. preconditions constraining the enabledness of a transition.

- The **in** part gives input port names (or sets of them if preceded by **multi**) *I_pn*, each in conjunction with a list *I_msgs* of message patterns expected to be present in the corresponding input buffer(s). When an ISM executes a transition, any free variables in message patterns are bound to the actual values that have been input. Each port names should appear at most once within a **in** part. Any input port not explicitly mentioned is left untouched.

- The **out** part gives output port names *O_pn*, each in conjunction with an expression *O_msgs* denoting a list of values designated for output to the corresponding port. The variant using **multi** is used to specify multicasts.

[3]These need not be constant but may contain also variables, which is useful for modeling generic transitions. In this case, one such transition has to be represented by a set of transitions within AutoFocus.

Each port name should be used at most once within each **out** part. Any output port not mentioned does not obtain new output.

- The **cmd** part gives the ISM command *cmd_expr* associated with the current transition. Such a command can be given in each transition if the **commands** subsection is present.

- The **post** part describes assignments of values *expr* to the local variables *lvar_name* of the data state. Variables not mentioned remain invariant. Alternatively, an expression *ds_expr'* may be given that represents the entire new data state after the transition. Assignments to the local variables suit an operational style, whereas an axiomatic style can be achieved using *ds_expr'* (in conjunction with suitable constraints in the preconditions).

4.2 Specification

Let cmd_type' denote cmd_type if given, otherwise the trivial type $unit$. Let st_type denote the local state type, that is the Cartesian product of the control state and the data state type ($ctrst\_type \times data\_st\_type$) if both types exist, otherwise only the existing control or data state type. Let $state\_type'$ denote $state\_type$ if given, otherwise st_type.

The **ism** theory section will add the following constants and definitions to the theory.

$name$`.transs`**:** Constant that is an inductively defined set of type
$param\_type_1 \Rightarrow \ \ldots \ \Rightarrow param\_type_n \Rightarrow$
$(cmd\_type', pn\_type, msg\_type, state\_type')$ `trans set`.

$name$`.transs.intros`**:** List of introduction rules defining $name$`.transs`.

$name$`.transs.`tr_name**:** The single introduction rule corresponding to tr_name, defined as follows.

- Let $bool\_expr_1, \ \ldots, \ bool\_expr_k$ denote the k preconditions of the rule (k may be 0).
- Let i be the empty message family $¤$ if no inputs are given. Otherwise for inputs $I\_msgs_1, \ \ldots, I\_msgs_m$ on ports $I\_pn_1, \ \ldots, I\_pn_m$ let i be the expression $¤(I\_pn_1 := I\_msgs_1, \ \ldots, I\_pn_m := I\_msgs_m)$

- Let o be the empty message family $¤$ if no outputs are given. Otherwise for outputs $O\_msgs_1, \ \ldots, O\_msgs_m$ on ports $O\_pn_1, \ \ldots, O\_pn_m$ let o be the expression $¤(O\_pn_1 := O\_msgs_1, \ \ldots, O\_pn_m := O\_msgs_m)$

- σ denotes the state of type $state\_type'$ before the transition, as follows. Let ds_name' be ds_name if given, otherwise s. Let st be the pair (cs_expr, ds_name') if both the control and data state are given, otherwise cs_expr or ds_name', depending on which part exists. If the $state\_type$ is not given, σ equals st. Otherwise, let $constr$ denote the[4] constructor for $state\_type'$ with a single argument of type st_type. Then σ equals $constr\ st$.

[4]Unless exactly one such constructor is found, an error occurs.

- σ' is defined like σ with the difference that cs_expr is replaced by cs_expr' and the data state may be modified (if a postcondition is given), as follows. If ds_expr' is given, ds_name' is replaced by this expression. Otherwise, let $lvar\_name_1,\ \ldots,\ lvar\_name_n$ denote the field names of updated fields in the data record, $expr_1,\ \ldots,\ expr_n$ the update expressions; then the data state will be modified to: $ds_name'(\!|lvar\_name_1 := expr_1,\ \ldots,\ lvar\_name_n := expr_n|\!)$

- Let cmd be cmd_expr if given, otherwise cmd_expr'.

The transition rule can then be written as:
$[\![bool\_expr_1;\ \ldots;\ bool\_expr_k]\!] \Longrightarrow$
$((i,\ \sigma),\ cmd,\ (o,\ \sigma')) \in name.\texttt{transss}\ p_1\ \ldots\ p_n$

name.`elims`: Elimination rule for *name*.`transs` as would be generated by an **inductive_cases** section. Note that this is the same as *name*.`transs.elims` but simplified according to the current simpset.

name.`ism`: Constant of type $param\_type_1 \Rightarrow\ \ldots\ \Rightarrow param\_type_n \Rightarrow$ $(cmd\_type', pn\_type, msg\_type, state\_type')$ `ism` that represents the ISM.

name.`ism_def`: Definition of the constant *name*.`ism`. Let st_expr0 denote the pair of the initial control and data state (cs_expr0, ds_expr0) in case both exist, otherwise the existing initial state (either cs_expr0 or ds_expr0). The definition looks like this:

$name.\texttt{ism}\ p_1\ \ldots\ p_n \equiv (\!|\texttt{inputs = } I\_pns\texttt{, outputs = } O\_pns\texttt{,}$
$\texttt{init = } st\_expr0\texttt{, trans = } name.\texttt{transs}\ p_1\ \ldots\ p_n|\!)$

Besides new definitions and constants for each ISM the following syntax translations will be added internally.

syntax (**symbols**)
  $name$`_transs_syntax` :: $param\_type_1 \Rightarrow\ \ldots\ \Rightarrow param\_type_n \Rightarrow$
    (pn_type, msg_type) `msgss` $\Rightarrow (pn\_type, msg\_type)$ `msgss` $\Rightarrow$
    $cmd\_type \Rightarrow state\_type' \Rightarrow state\_type' \Rightarrow$ `bool`
  $(name{:}\_{:}\ldots\_{:}\langle\_,\_\rangle\ -\_\rightarrow\ \langle\_,\_\rangle$ `[10, ..., 10, 10, 10, 10, 10, 10] 60)`
translations
  $name{:}p_1{:}\ldots p_n{:}\langle i, s\rangle\ -c\rightarrow\ \langle o, s'\rangle \rightleftharpoons$
  $((i,\ constr\ s),\ c,\ (o,\ constr\ s')) \in name.\texttt{transs}\ p_1\ \ldots\ p_n$

where *constr* denotes the constructor (if any) as defined above.

4.3 Accessing Constants and Definitions

We present here an example that is kept as simple as possible just to explain the access to the structures specified in Section 4.2.
theory *Example = ISM_package:*

We define a datatype port representing the ports of the ISM.

datatype *port = p1 | p2*

Also the state is simple: two control states $c1$, $c2$ and the data state consists of a record with two fields of type `nat`.

```
datatype A_control = c1 | c2
record A_data =
  f1 :: nat
  f2 :: nat
types A_state = "A_control × A_data"
```

Only two messages can be sent over the port.

```
datatype message = m1 | m2
```

The ISM A has one input and one output port. If A is in its initial state the control state is $c1$ and both local variables $f1$, $f2$ are set to 0. There is only one transition named $t1$: if in control state $c1$, the variables are both 0, and the message $m1$ is received on port $p1$, the variables will be set to 1 and the message $m2$ is sent on the output port $p2$. After the transition the control state is $c2$.

```
ism A =
  ports port
    inputs "{p1}"
    outputs "{p2}"
  messages message
  states
    control c1 :: A_control
    data "(|f1 = 0, f2 = 0|)", s :: A_data
  transitions
    t1: c1 → c2
      pre "f1 s = 0", "f2 s = 0"
      in p1 "[m1]"
      out p2 "[m2]"
      post f1 := 1, f2 := 1
```

The following constants are now defined.

```
term A.ism
term A.transs
```

The definition of the ISM A can be accessed as `A.ism_def`.

```
thm A.ism_def
```

```
"ism ≡ (|inputs = {p1}, outputs = {p2},
  init = (c1, (|f1 = 0, f2 = 0|)), trans = transs|)"
```

Also the defined rule can be accessed. All other theorems defined in normal inductive definitions (e.g. `A.transs.intros`) are of course also available.

```
thm A.transs.t1
```

```
"[[f1 ?s = 0; f2 ?s = 0]] ⟹ ((¤(p1 := [m0]) .@. ?i, c1, ?s),
  ¤(p2 := [m2]), ?i, c2, ?s(|f1 := 1, f2 := 1|)) ∈ transs"
```

```
end
```

References

[BLS00] Peter Braun, Heiko Lötzbeyer, and Oscar Slotosch. *Quest Users Guide*. TU München, March 2000.

[BW02] Stefan Berghofer and Markus Wenzel. *The Isabelle System Manual*, 2002. http://isabelle.in.tum.de/doc/system.pdf.

[Nan02] Sebastian Nanz. Integration of CASE tools and theorem provers: a framework for system modeling and verification with AutoFocus and Isabelle. Master's thesis, TU München, 2002. http://home.in.tum.de/nanz/csthesis/.

[Ohe02] David von Oheimb. Interacting State Machines: *a stateful approach to proving security*. In Ali Abdallah, Peter Ryan, and Steve Schneider, editors, *Proceedings from the BCS-FACS International Conference on Formal Aspects of Security 2002*, volume 2629 of *LNCS*. Springer-Verlag, 2002. http://ddvo.net/papers/ISMs.html.

[Pau02] Lawrence C. Paulson. *The Isabelle Reference Manual*, 2002. http://isabelle.in.tum.de/doc/ref.pdf.