Seal versus the ToolBus

P. Olivier

August 11, 1994

Contents

1 Introduction **3**

2 ASF+SDF Meta-environment **4**

3 SEAL **4**
3.1 SEAL objectives 4
3.2 GSE 4
3.3 Syntax 5
3.4 Semantics 5

4 ToolBus **6**
4.1 ToolBus architecture 6
4.2 Syntax 6
4.3 Semantics 8
4.4 Atomic actions 8

5 Translation of SEAL scripts to ToolBus scripts **9**
5.1 Introduction 9
5.2 ToolBus configuration 9
5.2.1 GSE tools: gse(File) 10
5.2.2 User interface tool: ui(File) 12
5.2.3 Specifications 14
5.2.4 Variable store tool: store 14
5.2.5 Boolean tool: boolean 15
5.2.6 Output tool 15

6 SEAL to ToolBus translation: recursive scheme **15**
6.1 An example translation 16
6.2 Connecting SEAL conditions and actions 18
6.3 Translating SEAL conditions 18
6.4 SEAL actions 22
6.5 Other UI objects 26
6.6 Optimization 27

7 SEAL to ToolBus translation: iterative scheme **28**
7.1 New tool for calculating SEAL conditions 28
7.2 An example translation 29
7.3 Start-stop button 31
7.4 Active tool 32

8 A prototype implementation of the iterative method **32**
8.1 Coupling with ASF+SDF 33
8.2 BPA example 34
8.3 Shortcomings 41

9 Conclusions **41**
9.1 Comparison of translation strategies 41
9.2 Writing ToolBus scripts by hand 41
9.3 Future Work 42

A SEAL syntax in SDF **43**

Bibliography **46**

Acknowledgements

Emotional assistance
First of all, I want to thank Paul Klint for providing the initial idea and inspiration for this project, and Wilco Koorn for providing assistance and background information whenever I needed it.

While working on this project for six months, I met a lot of interesting and above all friendly people, most of them within the context of the Programming Research Group. To avoid the risk of forgetting anyone, I like to thank them all. I especially want to express my appreciation for the members of the 'lunch brigade' who made me feel right at home from the start.

Last but not least, I want to thank Liang Hwie Oei, with whom I shared the office. We had many interesting discussions, both technical and philosophical. I hope you will be very happy and successful in Indonesia.

Technical assistance
Throughout this project, I have been able to reuse large pieces of software, and I would like to thank those who wrote these pieces for the time and energy thus saved. The two compilers I wrote in ASF+SDF are largely based on the SEAL to LISP compiler Wilco Koorn wrote. The prototype system, described in Chapter 8, makes extensive use of the ToolBus library written by Paul Klint ([BK94]), the SDF2lex/yacc converter of Wilco Koorn, the GEL pretty printer written by Job Ganzevoort and the ASF2C tool written by Pum Walters and Jasper Kamperman ([KW93]).

1 Introduction

In this thesis we will investigate the future of the Semantics-directed Environment Adaptation Language (SEAL) [Koo92] from the perspective of the ToolBus software application architecture. SEAL was developed to connect user interface objects to the ASF+SDF (Algebraic Specification Formalism plus Syntax Definition Formalism) [Kli93a] Meta-environment. The ASF+SDF Meta-environment is a generator for interactive programming environments. A generated programming environment, including the ASF+SDF Meta-environment itself, consists of a set of automatically generated tools like an editor and a parser cooperating with a set of computational components specified by the user in ASF+SDF. Typical components are typecheckers, compilers and interpreters. The complexity of the resulting system is very high. An attempt to get a grip on this complexity can be found in [SvVvW93]. Although very educational, the attempt is ad hoc and not very flexible. A more promising approach is taken in [Kli93b]. In that report, Paul Klint proposes a new software application architecture, the ToolBus, which is described in full detail in [BK94]. The ToolBus is a general communication structure based on Basic Process Algebra with iteration (BPA*)[BW90], instantiated with a suitable set of communication primitives.

2 ASF+SDF Meta-environment

SEAL was developed to control the user-interface aspects of the ASF+SDF Meta-environment. The ASF+SDF Meta-environment is a programming environment generator, which is capable of generating a programming environment from a formal definition of a language. ASF+SDF is an acronym for the combination of the Algebraic Specification Formalism and the Syntax Definition Formalism. These two formalisms are combined to make it possible to specify both the syntax and semantics of a language. The main distinctive feature of the ASF+SDF Meta-environment is the integration of the development environment and the generated environment into one application. This means you can edit both the language definition and the programs written in the language at the same time. In fact the same editor (GSE, see Section 3.2) is used to edit both.

3 SEAL

3.1 SEAL objectives

The specification of a computational component is one thing, integrating it with the user-interface is another. SEAL was developed to add user-interface objects, like buttons and menus, to the editor instances using a certain language and to connect these objects with functions offered by the computational component as specified in ASF+SDF.

The primary goal of SEAL is ensuring the uniformity of the user-interface. All interactions with the user are performed through editor instances, which enforces the primary goal. SEAL offers a fixed set of user-interface objects which can be connected to the computational component by a *SEAL script*. The "look and feel" of these objects is predefined to achieve the desired uniformity. The objects are placed at a predefined position in the editors window. This means you can concentrate on the semantics and forget all about the presentation aspects.

Another important goal of SEAL is the complete separation between the development of the computational component and the user-interface component. The application only knows about the existence of a user-interface. It does not know which objects are involved or how they depend on its state. The application merely reports a state change to the run-time part of the SEAL system.

3.2 GSE

A central part of the user interface of the ASF+SDF Meta-environment is the editor used to edit both language definitions and programs, GSE. GSE stands for Generic Structure Editor. It is generic because it is parameterized by a language definition. GSE is a hybrid editor, a combination of a structural editor and a text editor.

Communications between GSE and SEAL are mainly based on abstract syntax trees. SEAL can be used to retrieve (sub-)trees and replace them with others. An important concept in the editing model of GSE is the so called *focus* which designates a subtree. Text editing is only allowed inside the focus. SEAL provides commands to navigate the current focus. A subtree can contain a *meta-variable*, a placeholder for a term that has not been filled in yet.

3.3 Syntax

We will illustrate the SEAL syntax using a small SEAL script:

```
Configuration for language L is

button Typecheck
when focus root is Program
enable
  Prog := focus root;
  create("Typecheckerrors", L-tc : typecheck(Prog))
doc: "Simple typecheck button"
```

In this example, a button named ‘Typecheck’ is defined, which is added to all editor instances for language ‘L’. The button is enabled whenever the root tree in the editor instance to which this button is attached, is of type ‘Program’. When the button is enabled and the user presses it, the program residing in the editor instance is typechecked and the result is presented to the user in a new editor instance. A complete description of the SEAL syntax is found in appendix A.

3.4 Semantics

SEAL was designed to connect certain functions to user interface objects, and to determine the enabling conditions of these objects. This is done by supplying one or more `<condition, actions>` pairs for every UI object. When one of its conditions evaluates to true, a UI object is enabled, otherwise it is disabled. When a UI object is enabled and the user activates it, the actions corresponding to the first condition that evaluated to true are executed.

The conditions consist of a number of atomic conditions combined by the boolean operators *and*, *or* and *not*. The atomic conditions are:

- check the sort of a subtree residing in an editor instance.
- check whether the subtree is a meta variable.
- check if the subtree matches some string.

- call some arbitrary function in a specified module. The result of such a function call is interpreted as either *true*, when the result is one of "true", "True", "t" or "T", and *false* otherwise.

The actions fall into seven categories that perform the following tasks:

- Focus assignment: The current focus is replaced by a calculated term.
- Variable assignment: A variable is assigned a new value.
- Focus movement: The current focus is moved.
- While loop: a simple iteration construct.
- Editor creation: A new editor is created for text input or term output.
- Select statement: Let the user pick a (sub-)term from an editor.
- Function call: A function from a certain module is called, and the output is shown in a special output window.

4 ToolBus

4.1 ToolBus architecture

Although the ToolBus architecture is not yet stable, in this section we describe the version on which most of this thesis is based. Most of the features mentioned here will probably be present in the "final" ToolBus version.

One of the main objectives of the ToolBus is to provide a simple yet effective way to model the interaction between a number of tools. The interaction is modeled by a process script, representing a number of processes. A process can communicate with other processes and with the tools connected to the ToolBus. Figure 1 shows an instantiation of the ToolBus.

4.2 Syntax

The syntax of a ToolBus script is as follows[1] :

```
imports Integers

exports
  sorts ID VAR TERM TERM-LIST STRING ATOM ATOMIC-FUN PROC PNAME
  PROC-DEF FORMALS TB-CONFIG TB-SCRIPT

  lexical syntax
```

[1] For readers not acquainted with the SDF syntax, we hope it is self-explanatory. for a thorough description we refer to [HHKR89].

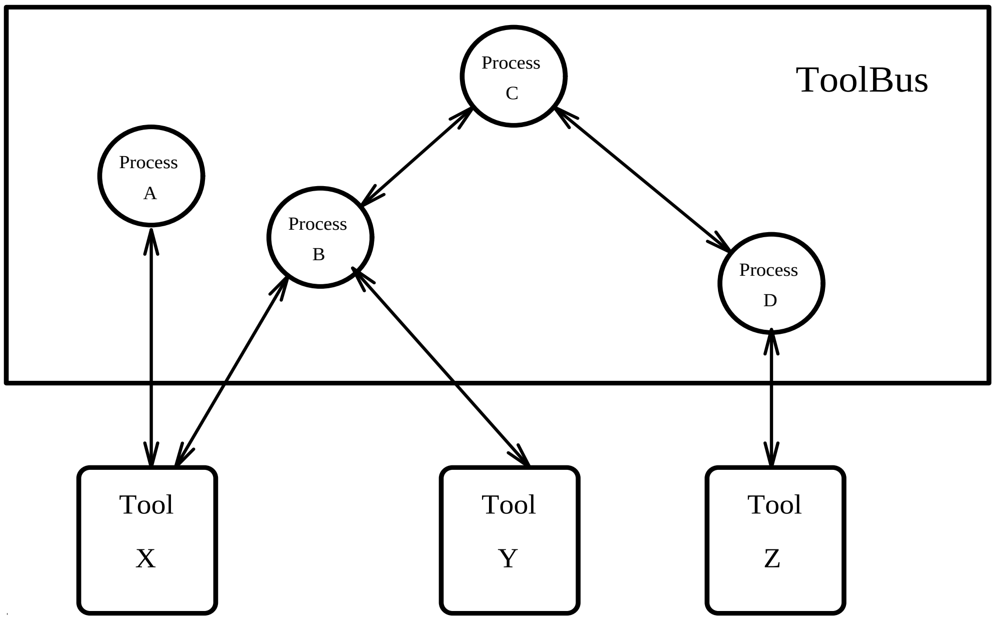

Figure 1: A ToolBus configuration with four processes and three tools

```
      [ \t\n]                              -> LAYOUT
      "%%" ~[\n]                           -> LAYOUT
      [a-z][A-Za-z0-9\-]*                  -> ID
      [A-Z][A-Za-z0-9\-]*                  -> PNAME
      [A-Z][A-Za-z0-9\-]*                  -> PNAME
      "\"" ~[\"]* "\""                     -> STRING
      [A-Z][A-Za-z0-9]*                    -> VAR

context-free syntax
      INT                                   -> TERM
      STRING                                -> TERM
      VAR                                   -> TERM
      ID                                    -> TERM
      ID "(" TERM-LIST ")"                  -> TERM
      {TERM ","}+                           -> TERM-LIST

      snd-msg                              -> ATOMIC-FUN
      rec-msg                              -> ATOMIC-FUN
      snd-note                             -> ATOMIC-FUN
      rec-note                             -> ATOMIC-FUN
      no-note                              -> ATOMIC-FUN
      rec-event                            -> ATOMIC-FUN
      snd-ack-event                        -> ATOMIC-FUN
      snd-eval                             -> ATOMIC-FUN
      snd-do                               -> ATOMIC-FUN
      rec-value                            -> ATOMIC-FUN
      subscribe                            -> ATOMIC-FUN
      unsubscribe                          -> ATOMIC-FUN
```

```
        ATOMIC-FUN ( TERM-LIST )        -> ATOM
        create ( PNAME )                -> ATOM
        create ( PNAME , TERM-LIST )    -> ATOM

        ATOM                                    -> PROC
        delta                                   -> PROC
        PROC "+" PROC                           -> PROC  {left}
        PROC "." PROC                           -> PROC  {right}
        PROC "*" PROC                           -> PROC  {left}
        "(" PROC ")"                            -> PROC  {bracket}
        PNAME                                   -> PROC
        PNAME ( TERM-LIST )                     -> PROC

        "(" {VAR ","}* ")"                      -> FORMALS
                                                -> FORMALS

        define PNAME FORMALS "=" PROC           -> PROC-DEF
        toolbus({PNAME ","}+ )                  -> TB-CONFIG
        PROC-DEF+ TB-CONFIG                     -> TB-SCRIPT

 priorities
        PROC "*" PROC -> PROC > "." >  PROC "+" PROC -> PROC
```

4.3 Semantics

ToolBus processes are based on BPA*. The "+" operator models nondeterministic choice, the "." operator models sequential composition, and the "*" operator models iteration.

It can be shown quite easily that the operator does not increase the expressive power of BPA:

$$P = x * y \Rightarrow P = y + x.(x * y) \Rightarrow P = y + x.P$$

So a process which uses $x * y$ can always be rewritten to a process containing the operators "+" and "." using recursion.

`toolbus(PROC1, ..., PROCn)`

which can, in a first approximation, be read as $PROC1 \| \ldots \| PROCn$.

4.4 Atomic actions

A ToolBus script can consist of the following atomic actions:

- snd-msg - Send a message to another process (synchronous communication)
- rec-msg - receive a message from another process

- snd-note - put a note in the note queue of another process (asynchronous communication)
- rec-note - fetch a note from the private note queue
- no-note - make sure no notes are waiting in the private note queue
- rec-event - receive an event from a tool
- snd-ack-event - acknowledge an event
- snd-eval - send a list of terms to a tool for evaluation
- rec-value - receive a value from a tool
- subscribe - subscribe to certain note types
- unsubscribe - unsubscribe from certain note types
- create - create a new process that executes in parallel with this process
- delta - deadlock, do not perform any more actions from this process

5 Translation of SEAL scripts to ToolBus scripts

5.1 Introduction

One way to compare the power of two (programming) formalisms is to define formalism B in terms of A. If you succeed, you know A is at least as powerful as B. We compare SEAL to the ToolBus. In particular, we want to know if the ToolBus is more powerful than SEAL to establish that a migration is possible. To assess the strong and weak points of the ToolBus, we researched two different translation schemes, a recursive scheme, and an iterative scheme. In the recursive scheme, most constructs in SEAL are translated as subprocesses in the ToolBus, yielding a difficult to read ToolBus script. In the iterative scheme, the behaviour of every button defined in SEAL is translated into one coherent ToolBus process.

5.2 ToolBus configuration

To make an effective comparison between SEAL scripts and ToolBus scripts, we have to come up with a ToolBus configuration that provides roughly the same functionality as the existing SEAL runtime system. As a consequence we have to give a set of tools (and the services they provide) similar to the current components used in the ASF+SDF Meta-environment.

An example of the use of these tools is depicted in Figure 2. In the following sections we describe all such tools and the services they provide. To this end we use the following format:

Usage: Shows how a ToolBus process can use this service.
Result: The result, if any, returned by the tool
Description: A short description of the service.
SEAL: States the SEAL equivalent (if any) of this service.

5.2.1 GSE tools: gse(File)

The *gse* tool represents the class of GSE instances. A GSE instance can be identified by the name of the file residing in the instance. We will use the term *gse(File)* to identify them, where *File* is the name of the file. Alternatively we will use *gse* when no actual instance is involved, for instance when we want to create a new editor instance.

The behaviour of a tool can be split into two categories: the events it generates, and the services it provides. Of all SEAL-related tools, only the gse tool and the ui tool generate events. The only event generated by a gse tool is the *changed* event. Whenever a change has occurred in the editor, for instance because the user typed some characters, the gse tool generates such an event. The typical reaction would be to re-evaluate the conditions of a button and update its enabling status.

Services:

Usage: snd-eval(gse(File), get-tree(<moves>))
Result: rec-value(gse(File), <tree>)
Description: Retrieves the subtree indicated by <moves> residing in the editor.
SEAL: X := focus <moves>

Usage: snd-eval(gse(File),do-select(<sort>))
Result: rec-value(gse(File), <tree>)
Description: Let the user point at a subtree of sort <sort>
within the editor, and return the subtree.
SEAL: select(<editor-name>, <sort>)

Usage: snd-eval(gse(File), check-sort(<moves>, <sort>)
Result: rec-value(gse(File), <boolean>)
Description: Check if the sort of a subtree designated by <moves>
has sort <sort>.
SEAL: <moves> is <sort> (in conditions)

Usage: snd-eval(gse(File), check-match(<moves>, <string>)
Result: rec-value(gse(File), <boolean>)
Description: Check if a subtree matches a certain tree pattern.
SEAL: <moves> matches <string>

Usage: snd-eval(gse(File), is-meta-var(<moves>)
Result: rec-value(gse(File), <boolean>)
Description: Check if a subtree is a meta variable.
SEAL: <moves> is-meta-var

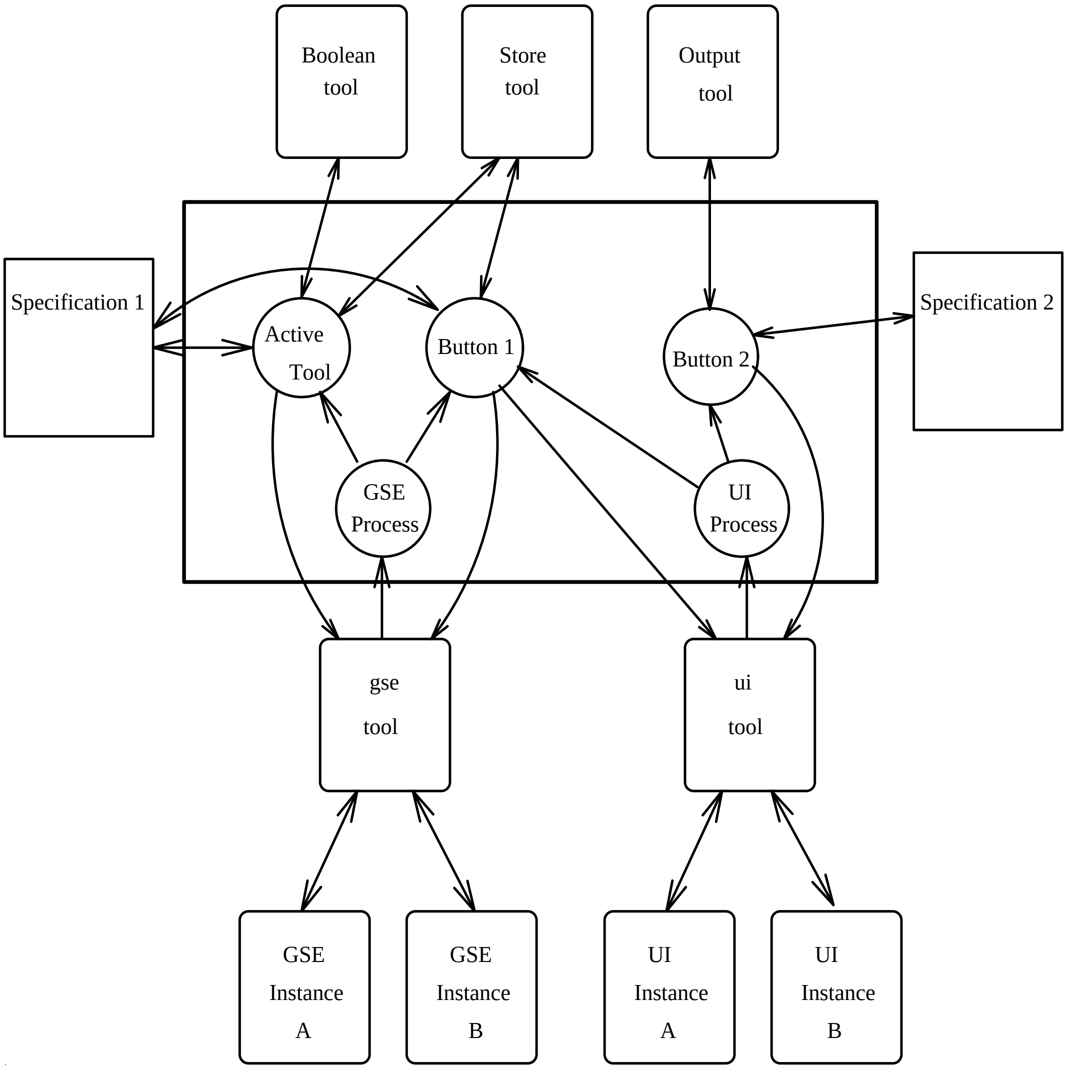

Figure 2: A useful ToolBus configuration

Usage: snd-do(gse(File), replace-focus(<moves>, <tree>))
Result:
Description: Replace the subtree indicated by **<moves>** by **<tree>**.
SEAL: focus <moves> := <tree>

Usage: snd-do(gse(File), move-focus(<moves>))
Result:
Description: Move the focus within an editor instance.
SEAL: focus <moves>

Usage: snd-do(gse, create-input(<name>, <module>, <text>, <sort>)
Result:
Description: create a editor instance to be used for text input.
SEAL: create(<name>, <module>, <text>, <sort>)

Usage: snd-do(gse, create-output(<name>, <term>)
Result:
Description: Create an editor instance to present **<term>** as output to the user.
SEAL: create(<name>, <term>)

Usage: snd-do(gse, destroy-editor(<name>))
Result:
Description: Destroy an editor instance.
SEAL: *not yet present in SEAL*

5.2.2 User interface tool: ui(File)

To present a number of user interface objects to the user, we introduce the *ui(File)* tool (or just *ui* when the name is not important), where *File* is the name of the file residing in the GSE instance to which this user interface tool is attached. Its main purpose is to keep track of the enabling status of a set of user interface objects connected to a GSE instance, and to generate events when the user activates one of these objects.

A UI object can either be *enabled* or *disabled*, depending on whether one of its enabling conditions evaluates to true or not. In addition, when it is enabled, we have to keep track of the enabling condition so we can execute the right set of actions when the user presses it, without re-evaluating the conditions. The only exception is the start-stop button. When such a button is enabled, it can either be activated or deactivated by the user.

A ui tool can generate two events:

- **pressed(ID, CR)**-event: A UI object with the unique identification "ID" has been pressed by the user. CR is the number of its enabling condition, and the appropriate actions must be executed.

- activated(ID, CR)-event: The start-stop button with the unique identification ID and enabling condition nr CR, has been activated by the user. The actions connected to this button have to be executed repeatedly, until the conditions fail or the user presses it again.

Notice the lack of a deactivated-event to go with the activated-event. We do not need a deactivated-event because when a start-stop button is activated, we use a polling strategy to determine when the user deactivates it (see the is-active service below.)

Services:

Usage: snd-do(ui(File), enable-object(<ui-object>, <cond-nr>))
Result:
Description: Enable a certain UI object,
and store the enabling condition number.
SEAL: button <ui-object>
enable

Usage: snd-do(ui(File), disable-object(<ui-object>, ID))
Result:
Description: Disable a certain UI object.
SEAL: *disabling of UI object after the conditions fail*

Usage: snd-eval(ui(File), is-active(<ui-object>))
Result: rec-value(ui(File), <boolean>)
Description: Check if a certain start-stop button is active. Note that this check should return false when the button is activated by the user, but disabled because none of its conditions evaluates to true.
SEAL: *the equivalent is part of the SEAL runtime system*

Usage: snd-eval(ui(File), is-enabled(<ui-object>)
Result: rec-value(ui(File), <boolean>)
Description: Check if a certain UI object is enabled or disabled.
SEAL: *SEAL runtime system*

Usage: snd-do(ui(File), disable-all) .
Result:
Description: Disable all UI objects in this user interface tool.
SEAL: *SEAL runtime system*

Usage: snd-do(ui, do-create(<name>))
Result:
Description: Add a user interface to go with the GSE instance <name>.
SEAL: *SEAL runtime system*

Usage: `snd-do(ui(File), add(<object>))`
Result:
Description: Add a user interface object like a button or menu-item. `<object>` contains an indication of the type and a unique identification, for instance, `button(Typecheck)`
SEAL: *SEAL runtime system*

Usage: `snd-do(UI(File), remove(<object>))`
Result:
Description: Remove a user interface object.
SEAL: *SEAL runtime system*

5.2.3 Specifications

Another class of tools are those tools that are specified in ASF+SDF. They are typically part of what is called "the computational component". For example a typechecker, compiler or evaluator. The services they provide vary from tool to tool: **Services:**

Usage: `snd-eval(<mod>, <proc-id>, <input-term>)`
Result: `rec-value(<mod>, <proc-id>, <output-term>)`
Description: Call a user specified function in module `<mod>` `<proc-id>` is an identification of the sending process.
SEAL: `X := <mod>:<input-term>`

5.2.4 Variable store tool: store

Variables can be used in ToolBus scripts, however an assignment operator is not provided. To model SEAL variables we need a tool that keeps track of the current value of a variable.

Services:

Usage: `snd-do(store, put("<variable>", <value>))`
Result:
Description: Assign a value to a variable.
SEAL: `<variable> := <value>`

Usage: `snd-eval(store, get("<variable>"))`
Result: `rec-value(store, <value>)`
Description: Retrieve the current value of a variable.
SEAL: Any usage of `<variable>` other than on the left hand side of an assignment.

Usage: snd-do(store, assign("<var1>", "<var2>"))
Result:
Description: Assign the value of one variable to another.
SEAL: <var1> := <var2>

5.2.5 Boolean tool: boolean

A less obvious tool needed to translate SEAL conditions, is a special purpose *interpreter* that interprets the result of a reduction as either true or false (it "booleanizes" a term). To model the current behaviour of SEAL as closely as possible, such an *oracle* takes a tree of any signature as input, and delivers either true or false based on the string representation of the input term. If the string representation is either "t", "T", "true", or "True", true is returned, otherwise the result is false.

Services:

Usage: snd-eval(boolean, <term>)
Result: rec-value(boolean, <boolean>)
Description: Interpret a term as either true or false.
SEAL: *SEAL runtime system*

5.2.6 Output tool

SEAL has an easy way to display the result of a function call: just show it on the special output window from which the Meta-environment was started. To keep the ToolBus version compatible, we introduce an output tool, that just displays every term that is sent to it.

Services:

Usage: snd-do(output, <term>)
Result:
Description: Display <term> in a special window.
SEAL: <module>:<function>(<arguments>)

6 SEAL to ToolBus translation: recursive scheme

The definition of the ToolBus configuration and the description of the necessary tools in the previous section, now places us in a position to translate SEAL scripts into ToolBus scripts. A SEAL script basically consists of the definition of a number of different UI objects. We describe a single UI type, the *button*, and defer other types until Section 6.5

For each button, we create one ToolBus process that is controlling its enabledness. The definition of a button consists of a list of $\langle \mathit{condition}, \mathit{actions} \rangle$ pairs. We have to find suitable translations for all SEAL constructs found in the conditions and actions, and we have to glue them together.

6.1 An example translation

Before we present a description of the translation of SEAL constructs into ToolBus script fragments, we present a small example of a translation. Consider the same SEAL script as discussed in Section 3.3:

```
Configuration for language L is

button Typecheck
when focus root is Program
enable
  Prog := focus root;
  create("Typecheckerrors", L-tc : typecheck(Prog))
doc: "Simple typecheck button"
```

The corresponding ToolBus script as generated by our first SEAL to ToolBus compiler:

```
define TOPLEVEL = create(GSE,test) .
                  create(UI,test) .
                  create(ProcTypecheck,test)

define ProcTypecheck(File) =
  subscribe(ngse) .
  subscribe(nui) .
  PTypecheckPART-1(File)

define COND0(File) =
  snd-eval(gse(File), check-sort(move-root, sortProgram)) .
  ( rec-value(gse(File), true) . T0(File)
  +
    rec-value(gse(File), false) . F0(File)
  )

define F0(File) =
  snd-do(gse(File), please-disable(butTypecheck))

define T0(File) =
  snd-do(gse(File), please-enable(butTypecheck, 1))

define PTypecheckPART-1(File) =
  rec-note(nui(File), pressed(butTypecheck, 1)) .
  snd-eval(gse(File), get-focus-tree(move-root)) .
  rec-value(gse(File), VAL) .
  snd-do(store, put("Prog", VAL)) .
  snd-eval(store, get("Prog")) .
```

```
  rec-value(store,VARProg) .
  snd-eval(modL-tc, funtypecheck(VARProg)) .
  rec-value(VAL) .
  snd-do(gse, create-output("Typecheckerrors", VAL))
+
  PTypecheckPART-2(File)

define PTypecheckPART-2(File) =
  rec-note(ngse(File), changed) . COND0(File)

define GSE(File) =
  snd-do(gse, create-output(File, "")) .
  ( rec-event(gse(File), changed) .
    snd-do(ui(File), disable-all) .
    snd-note(ngse(File), changed) .
    snd-ack-event(gse(File))
  ) . GSE(File)

define UI(File) =
  snd-do(ui, do-create(File)) .
  (
    rec-event(ui(File), EVENT) .
    snd-note(nui(File), EVENT) .
    snd-ack-event(ui(File))
  ) . UI(File)
```

The ToolBus script consists of three processes created by TOPLEVEL. The process GSE(File) waits for events from the gse(File) tool, and sends out notes to all subscribing processes, in this case ProcTypecheck. The UI(File) process does the same for the ui(File) tool. Most of the work is done by the PROCTypeCheck(File), which controls the enabling status and the actions of the TypeCheck button. The subprocess COND0(File) evaluates the condition. PTypecheckPART-1(File) waits for the user to press the button and subsequently executes the actions. PTypecheckPART-2(File) waits until the text in the editor is changed and then re-evaluates the condition.

In the actions

```
snd-do(gse(File), enable-button(typecheck, 1))
rec-event(gse(File), button(typecheck, 1))
```

the "1" denotes the condition that is responsible for enabling the typecheck button. In this simple example, the typecheck button has only one `<condition,action>` pair. In general however, it is vital to keep track of the enabling condition of a button to be able to execute the right set of actions when the user presses the button.

Notice that the use of the store tool in this small example is completely unnecessary. We have included a separate section (see Section 6.6) that describes such (often trivial) optimizations.

6.2 Connecting SEAL conditions and actions

Suppose we have a number of $\langle condition, actions \rangle$ pairs: `button B`
`when` C_1
`enable` A_1
`when` C_2
`enable` A_2
`etc.`
Suppose we can find some valid ToolBus actions C' that set the state of button B to indicate the first condition that evaluates to true. Let A'_i be the translation of action A_i. The connection between conditions and actions of B is then defined by process B-enabling as follows:

```
B-enabling
= rec-note(ngse(File), changed) .  C'
+ rec-note(ngse(File), button(B, 1)) .  A'1
+ rec-note(ngse(File), button(B, 2)) .  A'2
etc.
```

Whenever a change in the editor occurs, for instance because the user types some characters, all conditions are re-evaluated until one is found to be true and the state of the UI objects is changed accordingly by the actions in C'. Whenever a button is pressed, the appropriate ToolBus code is executed.

The previous construction requires an answer to the following two questions:

- Suppose we have a number of SEAL conditions, $C_1, \ldots, C_n$, can we find ToolBus code that sets the state of the accompanying button to indicate the first condition that evaluates to true ?
- Suppose we have SEAL actions $A_1, ..., A_n$, can we find corresponding ToolBus code for each of them that performs the same actions ?

These questions are answered in the following two sections.

6.3 Translating SEAL conditions

Suppose we have a number of SEAL conditions, $C_1, \ldots, Cn$. A single condition consists of a number of *atomic conditions* like: `<focus-expr> is <sort>` connected by the boolean operators "and", "or" and "not". The construction of the translation algorithm is based on the following observation:
The translation of a (sub-)condition is always of the following form:

```
X . ( rec-value(Tool, true) . T
      +
      rec-value(Tool, false) . F
    )
```

where T is the branch taken when the condition evaluates to true, and F the branch taken when it evaluates to false. The fact that this observation holds for all atomic conditions will be shown at the end of this section. Here, we will show how to construct conditions that are connected with the boolean operators. In order to demonstrate our scheme, we will use two conditions A and B that will be translated into:

```
X . ( rec-value(Tool, true) . T0
      +
      rec-value(Tool, false) . F0
    )
```

and

```
Y . ( rec-value(Tool', true) . T1
      +
      rec-value(Tool', false) . F1
    )
```

respectively. We now present the translation scheme for each of the boolean operators mentioned:

- not A
  Translating "not A" amounts to switching the T and F branches and yields:

  ```
  X . ( rec-value(Tool, true) . F0
        +
        rec-value(Tool, false) . T0
      )
  ```

- A and B
  We first translate condition A. Its false branch is final: nothing in condition B can make up for A's failure. When A evaluates to true, the result of the whole condition depends on the result of B. Translating "A and B" yields therefore:

  ```
  X . ( rec-value(Tool, true) . Y .
           ( rec-value(Tool', true) . T2
             +
             rec-value(Tool', false) . F2
           )
        +
        rec-value(Tool, false) . F2
      )
  ```

  (where F2 are the actions taken when the whole condition fails and T2 are the actions taken when the condition succeeds)

- A or B
  We first translate condition A. Its true branch is final, the condition evaluates to true. When A evaluates to false, the result of the whole condition depends on the result of B. The translation of "A or B" is:

```
X . ( rec-value(Tool, true) . T2
      +
      rec-value(Tool, false) . Y .
         ( rec-value(Tool', true) . T2
           +
           rec-value(Tool', false) . F2
         )
    )
```

To make this construction a little bit more concrete, let us consider the translation of '(A or B) and C':

```
X . ( rec-value(ToolA, true) . Z .
         ( rec-value(ToolC, true) . T
           +
           rec-value(ToolC, false) . F
         )
    +
      rec-value(ToolA,false) . Y .
         ( rec-value(ToolB, true) . Z .
             ( rec-value(ToolC, true) . T
               +
               rec-value(ToolC, false) . F
             )
           +
           rec-value(ToolB, false) . F
         )
    )
```

This example shows that we have to be careful for an exponential explosion in the size of the ToolBus script. Introduction of subprocesses[2] solves this problem:

```
X . ( rec-value(ToolA, true) . T0
      +
      rec-value(ToolA, false) . F0
    )

define F0 =
  Y . ( rec-value(ToolB, true) . T1
        +
        rec-value(ToolB, false) . F1
      )

define T0 =
  Z . ( rec-value(ToolC, true) . T1
         +
        rec-value(ToolC, false) . F1
      )
```

[2]In the actual implementation, all subprocesses are parameterized with a filename indicating the current editor instance (See 6.1)

This process can be applied recursively, until we have translated the whole condition. This translation scheme is very generic, and can be used for the translating all SEAL conditions (including while conditions.)

After translating a condition, what should we do next? Notice that the translation always has two "exits", a true and a false exit, indicated by the unbound Tn and Fn ($T1$ and $F1$ in our example). When we reach the true exit, we have to set the state of the button to indicate the current condition. This is done by the following ToolBus actions:

```
define Tn = snd-do(gse(File), enable-object(<object-id>, i))
```

where i is the current condition number. When we reach the false exit, we start translating the following condition, in order to find the first condition that evaluates to true:

```
define Fn = translation of condition i+1
```

When we reach the false exit of the last condition, we want to disable the button:

```
Fn = snd-do(gse(File), disable-object(<object-id>))
```

To conclude this construction, we have to show that all atomic conditions can be translated (like we claimed earlier) into the format:

```
X . ( rec-value(Tool, true) . T
      +
      rec-value(Tool, false) . F
    )
```

SEAL features four kinds of atomic conditions:

- `<focus-expr> is <sort>`
  checking the sort of a focus expression. This is computed by the GSE tool as are the other three kinds, The translation becomes:

  ```
  snd-eval(gse(File), check-focus-sort(<focus-expr>, <sort>)) .
    ( rec-value(gse(File), true) . T
      +
      rec-value(gse(File), false) . F
    )
  ```

- `<focus-expr> matches <string>`
  checking whether a focus expression matches a string.

  ```
  snd-eval(gse(File), check-focus-match(<focus-expr>, <string>)) .
    ( rec-value(gse(File), true) . T
      +
      rec-value(gse(File), false) . F
    )
  ```

- <focus-expr> is-meta-var
  checking whether a focus expression is a meta variable.

```
snd-eval(gse(File), is-meta-var(<focus-expr>)) .
  ( rec-value(gse(File), true) . T
    +
    rec-value(gse(File), false) . F
  }
```

- <module> : <function> (<focus-expr-list>)
  calling a function, and interpreting the result. The translation of such a condition requires two steps. First we have to calculate the result of the term <function>({<focus-expr-list>}) in <module>, then we have to interpret the result using the Boolean tool presented in Section 5.2.5.

```
snd-eval(<module>,<function>(<focus-expr-list>)) .
rec-value(<module>,R) .
snd-eval(boolean, R) .
  ( rec-value(boolean, true) . T
    +
    rec-value(boolean, false) . F
  )
```

6.4 SEAL actions

The next and final step is to translate a block of SEAL actions into ToolBus actions. The nature of SEAL actions is sequential, we therefore connect the translation of the individual SEAL actions with the ToolBus '.' operator. This leaves us the task of finding translations for each individual SEAL action. SEAL actions can be divided into six categories. We present the translation for each of them below.

1. <focus> := <seal-term>
   This type of SEAL action comes in four flavours:

   - <focus> := <variable>
     Replace the subtree determined by <focus> with the contents of a variable. Just order the GSE instance to do this:

     ```
     snd-eval(store, get("<variable>")) . rec-value(store, VAL) .
     snd-do(gse(File), replace-focus(<focus>, VAL))
     ```

   - <focus> := select(<filename>, <sort>)
     Let the user choose a subtree from the editor instance indicated by <filename>, and replace the current focus with the subtree chosen:

     ```
     snd-eval(gse(<filename>), select(<sort>)) .
     rec-value(gse(<filename>), T) .
     snd-do(gse(File), replace-focus(<focus>, T))
     ```

- `<focus> := select(<variable>, <sort>)`
  Let the user choose a subtree from an editor instance identified by the value of `<variable>` and replace the current focus with the subtree chosen:

  ```
  snd-eval(store, get("<variable>")) . rec-value(store, NAME) .
  snd-do(gse(NAME), replace-focus(<focus>, T))
  ```

- `<focus> := <module>:<function>(<var-1>,...,<var-n>)`
  Replace the current focus with the result of a function call:

  ```
  snd-eval(store, get("<var-1>")) . rec-value(store, VAR1) .
     ...
  snd-eval(store, get("<var-n>")) . rec-value(store, VARn) .
  snd-eval(<module>, <function>(VAR1,...,VARn)) .
  rec-value(<module>, VAL) .
  snd-do(gse(File), replace-focus-tree(<focus>, VAL))
  ```

2. `<variable> := <seal-term>`
   The representation of variables in a ToolBus script yields some complications. In particular, the assignment of a value to a variable is done using the store tool. SEAL features the following assignments:

   - `<variable-1> := <variable-2>`
     Assign the value of a variable to another variable:

     ```
     snd-do(store, assign("<variable-2>", "<variable-1>"))
     ```

   - `<variable> := select(<filename>, <sort>)`
     Ask the user to point at a subtree of a certain sort:

     ```
     snd-eval(gse(<filename>), select(<sort>)) .
     rec-value(gse(<filename>), T) .
     snd-do(store, put("<variable>", T))
     ```

   - `<variable-1> := select(<variable-2>, <sort>)`
     Ask the user to point at a subtree of a certain sort, in an editor determined by `<variable-2>`:

     ```
     snd-eval(store, get("<variable-2>")) . rec-value(store, NAME) .
     snd-eval(gse(NAME), select(<sort>)) . rec-value(gse(NAME), T) .
     snd-do(store, put("<variable-1>", T))
     ```

   - `<variable> := <module>:<function>(<var-1>,...,<var-n>)`
     Assignment of the result of a certain function call to a variable. Perform the reduction, and create an alias to its result:

     ```
     snd-eval(store, get(<var-1>)) . rec-value(store, VAR-1) .
       ...                             ...
     snd-eval(store, get(<var-n>)) . rec-value(store, VAR-n) .
     snd-eval(<module>, <function>(VAR1,...,VARn)) .
     rec-value(<module>, VAL) .
     snd-do(store, put("<variable>", VAL))
     ```

- `<variable> := <focus> <moves>`
  Assignment of a certain part of the focus to a variable:

  ```
  snd-eval(gse(File), get-focus-tree(<focus>)) .
  rec-value(gse(File), VAR) .
  snd-do(store, put("<variable>"))
  ```

3. `<seal-focus-expression>`
   These actions move the current focus inside an editor instance.

   - `<focus-moves>` or `focus <focus-moves>`
     Move the focus inside the current editor instance:

     ```
     snd-do(gse(File), move-focus(<focus-moves>))
     ```

   - `<filename>.focus <focus-moves>`
     Move the focus in a named editor instance:

     ```
     snd-do(gse(<filename>), move-focus(<focus-moves>))
     ```

   - `<variable>.focus <focus-moves>`
     Move the focus in a calculated editor instance:

     ```
     snd-eval(store, get(<variable>)) .
     rec-value(store, NAME) .
     snd-do(gse(NAME), move-focus(<focus-moves>))
     ```

4. `while <while-condition> do <actions> od`
   The translation of the `<while-condition>` is similar to the translation of the SEAL conditions as described in Section 6.3. One difference is that in a while condition three more atomic conditions can appear. Their translation is presented below. Another difference is the way in which we handle the true and false branches when we have evaluated the complete condition. The false branch is non-existent in a while loop. We do not do anything when the while-condition fails. When the condition succeeds, we have to execute the translation of `<actions>`, after which we just start over. This is most easily accomplished by introducing recursion variables at the appropriate places:

   ```
   COND = X . ( rec-value(Tool, true) . TBRANCH
      +
        rec-value(Tool, false)
      )
   TBRANCH = Y . COND
   ```

   Where X denotes the translation of the condition, and Y denotes the translation of the actions. The translation of the three new atomic conditions is:

   - focus `<moves>`
     Check if a certain focus move is possible. Again, we use the GSE instance for this purpose

```
snd-eval(gse(File), can-move-focus(<moves>)) .
  ( rec-value(gse(File), true) . T
  +
    rec-value(gse(File), false) . F
  )
```

- `<filename>.focus <moves>`
  Check if a focus move is possible in a certain editor. The translation is similar to the one above:

  ```
  snd-eval(gse(<filename>), can-move-focus(<moves>)) .
    ( rec-value(gse(<filename>), true) . T
    +
      rec-value(gse(<filename>), false) . F
    )
  ```

- `<variable>.focus <moves>`
  Check if a focus move is possible in another editor, who's name is held in the variable `<variable>` We first retrieve the value of `<variable>` from the store tool and then we let the gse tool do the checking:

  ```
  snd-eval(store, get("<variable>")) . rec-value(store,NAME) .
  snd-eval(gse(NAME), can-move-focus(<moves>)) .
   ( rec-value(gse(NAME), true) . T
   +
     rec-value(gse(NAME), false) . F
   )
  ```

5. `<module>:<function>(<var-1>,...,<var-n>)` In SEAL, the result of the function call is displayed in a special output window. This translates to:

   ```
   snd-eval(store, get(<var-1>) . rec-value(store, VAR1) .
   ...             ...                  ...
   snd-eval(store, get(<var-n>) . rec-value(store, VARn) .
   snd-eval(<module>, <function>(VAR1,...,VARn) .
   rec-value(<module>, VAL) .
   snd-do(output, VAL)
   ```

6. create(...)
   In SEAL there are four versions of this command, which are translated as follows:

   - `create(<filename>, <module>, <text>, <sort>)`
     Create a new editor instance to be used for text-input:

     ```
     snd-do(gse, create-input(<filename>, <module>, <text>, <sort>)) .
     ```

   - `create(<variable>, <module>, <text>, <sort>)`
     Create a new editor instance with a calculated filename, to be used as a text-input window:

     ```
     snd-eval(store, get(<variable>)) . rec-value(store, NAME) .
     snd-do(gse, create-input(NAME, <module>, <text>, <sort>) .
     ```

- `create(<filename>, <variable>)`
  Create a new editor instance to be used as a term-output window:

  ```
  snd-eval(store, get(<variable>)) . rec-value(store, TERM) .
  snd-do(gse, create-output(<filename>, TERM)) .
  ```

- `create(<variable-1>, <variable-2>)`
  Create a new editor instance with a calculated filename, to be used as a term-output window:

  ```
  snd-eval(store, get("<variable-1>")) . rec-value(store, NAME) .
  snd-eval(store, get("<variable-2>")) . rec-value(store, VAL) .
  snd-do(gse, create-output(NAME, VAL)) .
  ```

6.5 Other UI objects

We have shown how we can translate a SEAL description of the UI object "button", but how can we translate descriptions of the other UI object types? SEAL knows about the following object types:

- button
- start-stop button
- menu
- menu-item
- active tool

Menu-items act just like buttons. They just look different but they act exactly alike. Menus are nothing but collections of menu-items. This leaves us with the start-stop buttons and the active tools. An active tool executes a certain block (which block depends on which condition evaluates to true) of actions every time a change in the editor occurs. Its translation looks like this (compare with section 6.2, T1 is the true branch of COND1, F1 the false branch of COND1 etc.):

```
AT = rec-note(ngse(File), changed) . COND1 . AT
T1 = ACT1
F1 = COND2
T2 = ACT2
F2 = COND3
 ...
Fn = AT
```

This more than cryptic description deserves some explanation. The basic idea is this: when a change in the editor occurs, we evaluate the first condition. If it evaluates to true, we execute the actions corresponding to the first condition, after which the change event is acknowledged. When the first condition evaluates to false, we check the second condition. If it succeeds we execute the actions that belong to it, else we check the third condition (if present). When we run out of conditions and all of them fail, we just start over.

When modeling the start-stop button, if the button is “off” we wait for the user to turn it “on”. When the button is “on” we repeatedly execute the actions corresponding to it, checking every time we execute an action whether the button is still “on”. The translation scheme for a start-stop button becomes:

```
SSB = rec-note(ngse(File), change) . C
      + rec-note(nui(File), pressed(B, 1)) . SSB1-1
      + rec-note(nui(File), pressed(B, 2)) . SSB2-1
      etc.

SSB1-1 = A1-1 . rec-eval(gse(File), is-active(<button-name>)) .
     ( rec-value(gse(File), true) . SSB1-2
     + rec-value(gse(File), false)
     )
SSB1-2 = A1-2 . rec-eval(gse(File), is-active(<button-name>)) .
     ( rec-value(gse(File), true) . SSB1-3
     + rec-value(gse(File), false)
     )
...                      ...

SSB1-n = A1-n . rec-eval(gse(File), is-active(<button-name>)) .
     ( rec-value(gse(File), true) . SSB1-1
     + rec-value(gse(File), false)
     )

SSB2-1 = A2' . rec-eval(gse(File), is-active(<button-name>)) .
       ( rec-value(gse(File), true) . SSB2-2
       + rec-value(gse(File), false)
       )
etc.
```

6.6 Optimization

The previous sections showed the feasibility of translating an arbitrary SEAL script to a ToolBus script, but some of these translations where quite ugly. Without diving to deep into the world of optimizations, we like to introduce an obvious improvement in the translation process. Take for instance the following SEAL actions:

```
VAR := focus
create("output", module : func(VAR));
```

The ToolBus translation of this pair of actions is:

```
snd-eval(module, func(focus)) . rec-val(module, TERM) .
snd-do(store, put("VAR", TERM)) .
snd-eval(store, get("VAR")) . rec-val(store, VAR) .
snd-do(gse, create-output("output", VAR))
```

A natural improvement would be to use the value of VAR directly, without intervention of the store tool:

```
snd-eval(module, func(focus)) . rec-val(module, VAR) .
snd-eval(gse, create-output("output", VAR))
```

In the next section we will present a translation technique that completely eliminates the need for the store tool.

7 SEAL to ToolBus translation: iterative scheme

The previous translation mechanism results in a set of (sub-) processes which, at first glance, seem to be randomly connected. We would rather capture the behaviour of a UI object in a single coherent process. In this section we will show that this goal can be met using the iteration construct found in the ToolBus, together with a new tool for evaluating SEAL conditions. The simplicity of the resulting ToolBus scripts has another advantage: we can completely eliminate the use of the store tool, if we eliminate the spurious variable to variable (VAR1 := VAR2) assignment from SEAL. With the first translation method the ui tool had to keep track of the enabling condition of the individual UI objects. In the new approach, this is no longer necessary. All the ui tool has to keep track of is the enabling or disabling of the UI objects (and the activation state of the start-stop buttons), which means another simplification.

7.1 New tool for calculating SEAL conditions

To circumvent the control flow complexity encountered when translating SEAL conditions, we let a "condition" tool calculate them. The evaluation of a condition always requires interaction(s) with other tools. We do not want the condition tool to directly access other tools. Some kind of interaction is needed between the ToolBus process and the condition tool, to make it possible for the process to contact other tools. Suppose we want to translate a SEAL condition, we first convert this condition to a prefixed form to adhere to the ToolBus convention about terms. When we send this term to the condition tool for evaluation, the condition tool will ask us to evaluate some terms and return the result to it. When we have supplied enough information for the condition tool to make its decision, it returns either true or false depending on the result of the condition:

```
snd-eval(condition(<name>, <proc-id>), evaluate(<Prefix condition>)) .
(
  rec-value(condition(<name>, <proc-id>), please-eval(TOOL, TERM)) .
  snd-eval(TOOL, <proc-id>, TERM) .
  rec-value(TOOL, <proc-id>, VAL) .
  snd-eval(condition(<name>, <proc-id>, VAL)
) * ( rec-value(condition(<name>, <proc-id>), true)
      +
      rec-value(condition(<name>, <proc-id>, false)
    )
```

Where `<Prefix condition>` is the SEAL condition translated into a ToolBus acceptable format, `<name>` stands for the name of the editor instance this process is attached to (needed to evaluate conditions like "is-meta-var"), and `<proc-id>` is a unique process identifier, for instance the name of the button. This process identifier is primarily used to avoid confusion when two processes communicate with the condition tool.
If we have a condition like:

```
when focus is-meta-var or focus root is Program
```

The translation would look like this:

```
snd-eval(condition(name, button),
 evaluate(or(is-meta-var(focus), check-sort(root, sortProgram)))) .
(
  rec-value(condition(name, button), please-eval(TOOL, TERM)) .
  snd-eval(TOOL, button, TERM) . rec-value(TOOL, button, VAL) .
  snd-eval(condition(name, button), VAL)
) * ( rec-value(condition(name, button), true)
      +
      rec-value(condition(name, button), false)
    )
```

"sort" is inserted before "Program" because else the ToolBus would see Program as a variable which it clearly is not.
A possible execution trace would be:

```
snd-eval(condition(name, button),
 evaluate(or(is-meta-var(focus), check-sort(root, sortProgram)))) .
rec-value(condition(name, button), please-eval(gse(name),is-meta-var(focus) .
snd-eval(gse(name), button, is-meta-var(focus)) .
rec-value(gse(name), button, true) .
snd-eval(condition(name, button), true) .
rec-value(condition(name, button), true)
```

7.2 An example translation

Consider the SEAL script presented in Section 6.1:

```
Configuration for language L is

button Typecheck
when focus root is Program
enable
  Prog := focus root;
  create("Typecheckerrors", L-tc : typecheck(Prog))
doc: "Simple typecheck button"
```

The corresponding ToolBus script, generated by our second SEAL to ToolBus compiler based on the iterative scheme:

```
process PROCTypecheck(Local) =
  subscribe(ngse) .
  subscribe(nui) .
  ( snd-eval(condition(Local, objTypecheck), do-eval(Local,
             check-sort(move-focus(move-root), sortProgram))) .
    ( rec-value(condition(Local, objTypecheck), eval(TOOL,TERM)) .
      snd-eval(TOOL, objTypecheck, TERM) .
      rec-value(TOOL, objTypecheck, VAL) .
      snd-eval(condition(Local, objTypecheck), VAL)
    ) * ( rec-value(condition(Local, objTypecheck) ,true) .
          snd-do(ui(Local), plz-enable(objTypecheck)) .
          ( rec-note(nui(Local), pressed(objTypecheck)) .
            ( snd-eval(gse(Local), get-focus-tree(move-root)) .
              rec-value(gse(Local), VARProg)
            ) .
            snd-eval(modL-tc, funtypecheck(VARProg)) .
            rec-value(modL-tc,VAL) .
            snd-do(em, create-output("Typecheckerrors", VAL))
          ) *
          rec-note(ngse(Local), changed)
        +
          rec-value(condition(Local, objTypecheck), false) .
          rec-note(ngse (Local), changed)
        )
  ) * delta

process GSE(Local) = ( rec-event(gse(Local), changed) .
                       snd-do(ui(Local), disable-all) .
                       snd-note(ngse(Local), changed) .
                       snd-ack-event(gse(Local))
                     ) * delta

process UI(Local)  = ( rec-event(ui(Local), EVENT) .
                       snd-note(nui(Local), EVENT) .
                       snd-ack-event(gse(Local))
                     ) * delta
```

The `GSE(File)` and `UI(File)` processes still only translate events into appropriate notes. The `PROCTypecheck(File)` process starts with a loop that evaluates the SEAL condition `when focus root is Program`, as explained above. This loop can end for two reasons: the condition can succeed or it can fail. If it succeeds, the process waits until the button is pressed (triggering the execution of the actions), or a change has occurred in the GSE instance, in which case we have to evaluate the conditions again. Notice that the translation of the actions is identical to the first translation method, except for the elimination of the store tool, and the translation of a while loop. The translation of a while loop looks like this:

```
  snd-eval(condition(<name>, <proc-id>), <prefixed-condition>) .
  ( rec-value(condition(<name>, <proc-id>), evaluate(TOOL, TERM)) .
```

```
      snd-eval(TOOL, <proc-id>, TERM) .
      rec-value(TOOL, <proc-id>, VAL) .
      snd-eval(condition(<name>, <proc-id>), VAL)
  ) * ( rec-value(condition(<name>, <proc-id>), true) .
        <while-body code> .
        snd-eval(condition(<name>, <proc-id>), <prefixed-condition>) .
        ( rec-value(condition(<name>, <proc-id>), evaluate(TOOL, TERM)) .
          snd-eval(TOOL, <proc-id>, TERM) .
          rec-value(TOOL, <proc-id>, VAL) .
          snd-eval(condition(<name>, <proc-id>), VAL)
        ) * ( rec-value(condition(<name>, <proc-id>), true) .
              snd-eval(condition(<name>, <proc-id>), true)
            +
              rec-value(condition(<name>, <proc-id>), false) .
              snd-eval(condition(<name>, <proc-id>), false)
            )
      ) * rec-value(condition(<name>, <proc-id>), false)
```

First, the condition is checked. When the condition evaluates to false, the while actions are not performed, and execution continues after the while. When the condition evaluates to true, a loop is entered that first executes the while-body code and then re-evaluates the condition.

7.3 Start-stop button

The skeleton of the translation of a start-stop button looks like this:

```
define PROCTest(File)   =
subscribe(ngse) .
subscribe(nui) .
( snd-eval(condition(<name>, <proc-id>), <prefixed-condition>) .
  ( rec-value(condition(<name>, <proc-id>), eval(TOOL,TERM)) .
    snd-eval(TOOL, <proc-id>, TERM) .
    rec-value(TOOL, <proc-id>, VAL) .
    snd-eval(condition(<name>, <proc-id>), VAL)
  ) * ( rec-value(condition(<name>, <proc-id>), true) .
        snd-do(ui(File), please-enable(objTest)) .
        ( rec-note(nui(File), pressed(objTest)) .
          snd-eval(ui(File), is-active(objTest)) .
          ( rec-value(ui(File), true)
            <action-1> .
            snd-eval(gse(File), is-active(objTest)) .
            ( rec-value(gse(File), true) .
              <action-2> .
              ...     ...
              +
              rec-value(gse(File), false)
            ) .
```

```
            snd-eval(ui(File), is-active(objTest))
          ) * rec-value(ui(File), false)
        ) * rec-note(ngse(File), changed)
      +
        rec-value(condition(<name>, <proc-id>),false) .
        rec-note(ngse(File), changed)
      )
) * delta
```

The trick behind the translation of start-stop buttons is to check the status of the button after every execution of an action. We only continue to execute the next action if the button is still activated. We keep repeating the actions until the button is deactivated. This deactivation can either be explicit deactivation by the user, or an implicit deactivation by the GSE(File) process mentioned in the previous section (7.2).

7.4 Active tool

The skeleton of the translation of an active tool looks like this:

```
define ActiveTool(File) =
  subscribe(ngse) .
  ( snd-eval(condition(<name>, <proc-id>), <prefixed-condition>) .
    ( rec-value(condition(<name>, <proc-id>), eval(TOOL,TERM)) .
      snd-eval(TOOL, <proc-id>, TERM) .
      rec-value(TOOL, <proc-id>, VAL) .
      snd-eval(condition(<name>, <proc-id>), VAL)
    ) * ( rec-value(condition(<name>, <proc-id>), true) .
          <translation of actions> .
          rec-note(ngse(File), changed)
        +
          rec-value(condition(<name>, <proc-id>), false) .
          rec-note(ngse(File), changed)
        )
  ) * delta
```

When the condition evaluates to true, we execute the actions once, and then wait until a change occurs in the editor. When it evaluates to false, we just wait for a change.

8 A prototype implementation of the iterative method

To gain some confidence in the second translation strategy (the most promising one) and the compiler specified in ASF+SDF, we implemented a prototype system. using the ToolBus prototype and the set of tools as described in Section 5.2

8.1 Coupling with ASF+SDF

The boolean tool and condition tool are written in C++, and the gse tool and ui tool are written in TCL. However, all but the most trivial SEAL scripts call functions in modules specified in ASF+SDF, so we had to find a way to use such modules. We

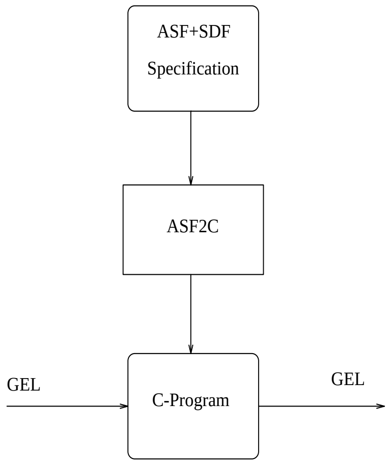

Figure 3: ASF2C tool

decided to use the ASF2C tool for this purpose. This tool takes a complete ASF+SDF specification as input, and creates a C program. This C program takes a GEL (Graph Exchange Language, see [Kam94]) text representing a term in the original specification as input, reduces it by the rules given in the specification, and writes the reduced output term as another GEL text (Figure 3).

The other tools are unable to cope directly with GEL. This means some conversion is necessary when transferring terms over the ToolBus. There are three ways to solve this problem:

- Send GEL over the ToolBus and use a GEL pretty printer to convert terms from GEL to plain character strings before sending them to tools that do not understand GEL. If the tool returns some string as result, this string has to be parsed and converted to GEL before sending it to the ToolBus (See Figure 4).
- Send plain character strings over the ToolBus and convert them to GEL before sending them to a tool generated by ASF2C. The result term has to be transformed into a character string using a GEL pretty printer.
- Send both GEL terms and plain character strings over the ToolBus and convert them whenever necessary by sending them to a pretty printer or parser tool (See Figure 6). This is the most efficient method, because the number of transformations can be kept as small as possible. A disadvantage is that the ToolBus scripts have to contain explicit sends and receives to the parser/pretty printer and are therefore more complex.

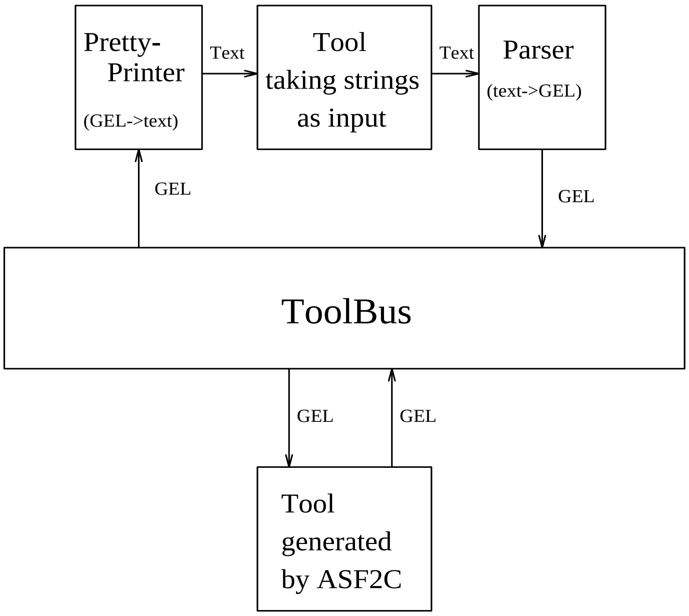

Figure 4: Sending GEL over the ToolBus

Because Paul Klint used the third approach before, and showed it was feasible, we decided to experiment with the second approach[3] (See Figure 5).

As a parser, we use a tool developed by Wilco Koorn that generates a lex and yacc compatible parser given an ASF+SDF module. The pretty printer is a general GEL pretty printer written by Job Ganzevoort.

8.2 BPA example

As an example, we implemented a small but usable system to manipulate Basic Process Algebra (BPA, [BW90]) terms. The basic idea is to make it easy for the user to transform BPA terms using buttons defined in SEAL. Possible transformations are based on the axioms of BPA:

$$x + y = y + x \tag{1}$$
$$(x + y) + z = x + (y + z) \tag{2}$$
$$x + x = x \tag{3}$$
$$(x + y).z = x.z + y.z \tag{4}$$
$$(x.y).z = x.(y.z) \tag{5}$$

We define five buttons using SEAL:

- (1) CommuChoice - transform using the commutativity of the + (choice) operator

[3] "to boldly go where no man has gone before"

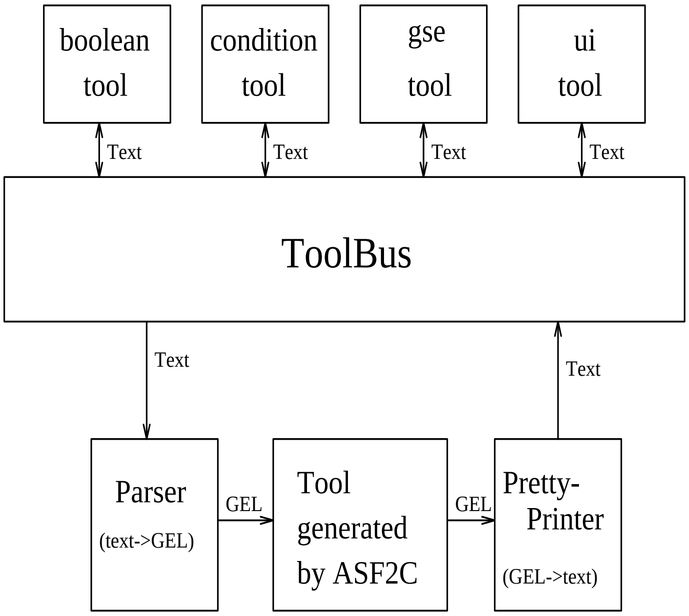

Figure 5: Sending text over the ToolBus

- (2) AssoChoice - transform using the associativity of the + operator
- (3) IdemChoice - transform using the idempotency of the + operator
- (4) Distribute - transform using the right distributivity of . over +
- (5) AssoSeq - transform using the associativity of the . (sequential) operator

All buttons replace the current focus with the result after the appropriate transformation. The SEAL script used to accomplish this is:

```
Configuration for language BPA is
button CommuChoice
when focus is PA-Term and BPA:can-Commu-Choice(focus)
enable
  FocusVar := focus;
  focus := BPA:commu-Choice(FocusVar)
doc: "Commutative +"

button AssoChoice
when focus is PA-Term and BPA:can-Asso-Choice(focus)
enable
  FocusVar := focus;
  focus := BPA:asso-Choice (FocusVar)
doc: "Associative +"
```

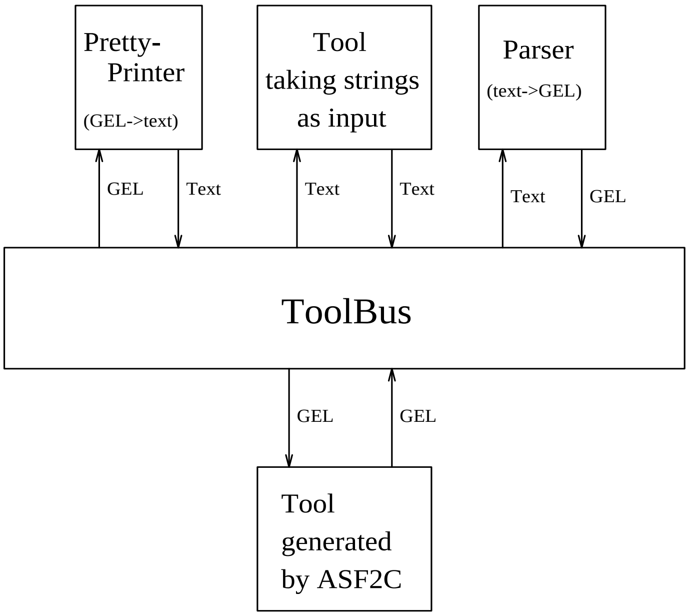

Figure 6: Sending both GEL and text over the ToolBus

```
button IdempChoice
when focus is PA-Term and BPA:can-Idem-Choice(focus)
enable
  FocusVar := focus;
  focus := BPA:idem-Choice(FocusVar)
doc: "Idempotent +"

button Distribute
when focus is PA-Term and BPA:can-Distribute(focus)
enable
  FocusVar := focus;
  focus := BPA:distribute(FocusVar)
doc: "Right distributivity of . over +"

button AssoSeq
when focus is PA-Term and BPA:can-Asso-Seq(focus)
enable
  FocusVar := focus;
  focus := BPA:asso-Seq(FocusVar)
doc: "Associative ."
```

The ASF+SDF module BPA exports the following functions (used in the SEAL script):

- can-Commu-Choice: check if axiom 1 (commutativity of choice operator) can be applied.
- can-Asso-Chocice: check if axiom 2 (associativity of choice operator) can be applied.
- can-Idem-Choice: check if axiom 3 (idempotency of choice operator) can be applied.
- can-Distribute: check if axiom 4 (right distributivity) can be applied.
- can-Asso-Seq: check if axiom 5 (associativity of sequence operator) can be applied.
- commu-Choice: actually apply axiom 1
- asso-Choice: apply axiom 2
- idem-Choice: apply axiom 3
- distribute: apply axiom 4
- asso-Seq: apply axiom 5

Using the current ASF+SDF Meta-environment, the result is show in Figure 7.

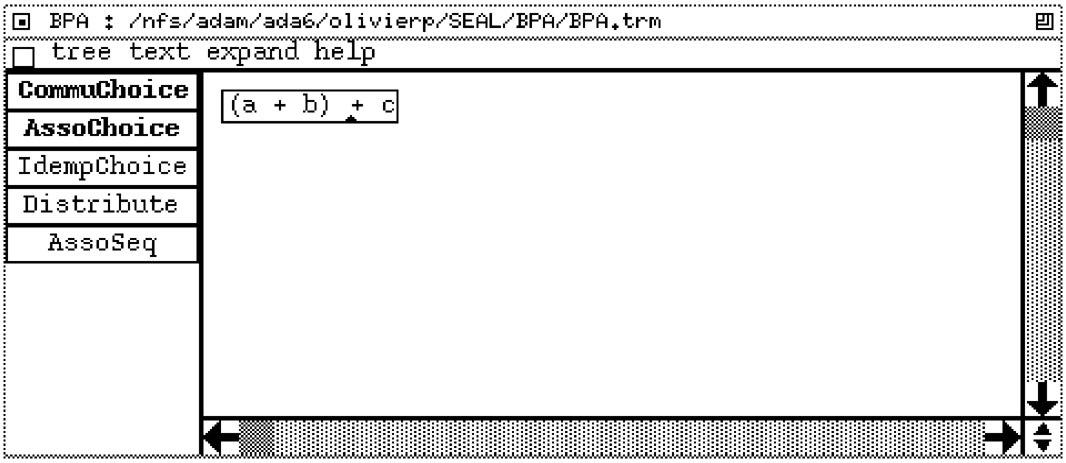

Figure 7: BPA under control of SEAL

To give an idea of the resulting ToolBus script after compilation, we present some fragments of it:

```
define TOPLEVEL = create(GSE,test) .
                  create(UI,test) .
                  create(CONTROLSBPA, test)

define PROCCommuChoice(Local) =
  subscribe(ngse) . subscribe(nui) .
  snd-do(ui(Local), add-button(objCommuChoice, "CommuChoice")) .
  ( snd-eval(gse(Local), get-tree(current-focus)) .
    rec-value(gse(Local), V20) .
    snd-eval(condition(Local, objCommuChoice), evaluate(
    b-and(check-sort(current-focus,sortPA-Term ),
          evaluate(modBPA(term), can-Commu-Choice(V20))))) .
    ( rec-value(condition(Local,objCommuChoice), please-eval(TOOL, TERM)) .
      snd-eval(TOOL,objCommuChoice,TERM) .
      rec-value(TOOL,objCommuChoice,VAL) .
      snd-eval(condition(Local, objCommuChoice), VAL)
    ) *
    ( rec-value(condition(Local, objCommuChoice), true) .
      snd-do(ui(Local), please-enable(objCommuChoice)) .
      ( rec-note(nui(Local), pressed(objCommuChoice)) .
        snd-eval(gse(Local), get-tree(current-focus)) .
        rec-value(gse(Local), VARFocusVar) .
        snd-eval(modBPA(term), commu-Choice(VARFocusVar)) .
        rec-value(modBPA(term), VAL) .
        snd-do(gse(Local), replace-focus(VAL))
      ) *
        rec-note(ngse(Local), changed)
      +
        rec-value(condition(Local, objCommuChoice), false) .
        snd-do(ui(Local), please-disable(objCommuChoice)) .
        rec-note(ngse(Local), changed)
    )
  ) * delta

...

define CONTROLSBPA(Local) =
  create(PROCCommuChoice,Local) . create(PROCAssoChoice,Local) .
  create(PROCIdempChoice,Local) . create(PROCDistribute,Local) .
  create(PROCAssoSeq,Local)

define GSE(Local) =
  snd-do(gse, create-output ( Local,"" )) .
  ( rec-event(gse (Local), changed) .
    snd-do(ui(Local), disable-all) .
    snd-note(ngse(Local), changed) .
    snd-ack-event(gse(Local))
  ) * delta
```

```
define UI(Local) =
  snd-do(ui, do-create(Local)) .
  ( rec-event(ui(Local), EVENT) .
    snd-note(nui(Local), EVENT) .
    snd-ack-event(ui(Local))
  ) * delta

tool gse       { command = "wish-adapter -name gse -script gse.tcl" }
tool ui        { command = "wish-adapter -name ui -script ui.tcl" }
tool condition { command = "condition" }
tool boolean   { command = "boolean" }
tool modBPA    { command =
  "asf2c-adapter -name modBPA -togel BPA2Gel -trs BPA -pretty default-pp" }

toolbus(TOPLEVEL)
```

The tool section, needed by the ToolBus simulator to find the tools, was added manually, but the compiler could easily be extended to handle this. This tool section basically defines which executable implements which tool. The modBPA tool is an instance of the generic asf2c-adapter we wrote. This adapter takes 4 parameters:

- -name: Name of the tool in question (in this case “modBPA”)
- -togel: Parser used to translate a term into GEL format (in our case “BPA2Gel”, created with the Sdf2yacc/lex tools written by Wilco Koorn)
- -trs: The name of the executable generated by ASF2C (“BPA”)
- -pretty: The name of the pretty printer used to transform the resulting GEL term back into a string representation (in our case the default pretty printer “default-pp”)

The asf2c-adapter can receive terms of the following form:

```
do-eval(modBPA(<type>), <process-id>, <term>)
```

With **<type>** the type of term, either *gel* or (in our case) *term*. **process-id>** is used to identify the sending process, so no mixups occur when two separate processes use the condition tool interleaved. **<term>** is the actual term that has to be evaluated. The type of the term is determined by the **<type>** field. Although in our examples we never transfer GEL over the ToolBus, the type field makes this possible for future experiments.

After receiving a valid term, the asf2c-adapter first creates a GEL representation of the input term using the program specified with the “-togel” parameter. This GEL term is fed to the executable specified with the “-trs” parameter. The resulting GEL term is passed to the pretty printer specified by the “-pretty” parameter. The resulting string is returned to the ToolBus.

A user session might be: initially all buttons are disabled because the editor is empty. When we enter the string “a+b” in the editor prototype, and press the

"Changed" button, the function "can-Commu-Choice(a+b)" will be evaluated by the asf2c-adapter and will return true. Because no notion of *sort* is known to the editor prototype, it asks the user if the current focus has the right sort (PA-Term in this case). If the user's answer is positive, the "CommuChoice" button will be enabled (See Figure 8). When the user presses this button, the function "commu-Choice(a+b)" is

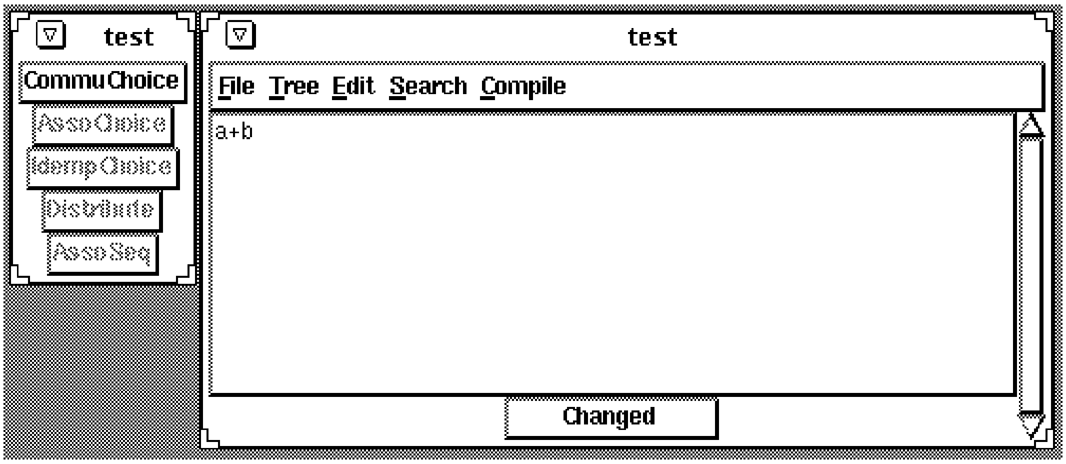

Figure 8: a+b enables "CommuChoice" button

4

evaluated by the asf2c-adapter and will return b+a, which is then placed in the editor (See Figure 9).

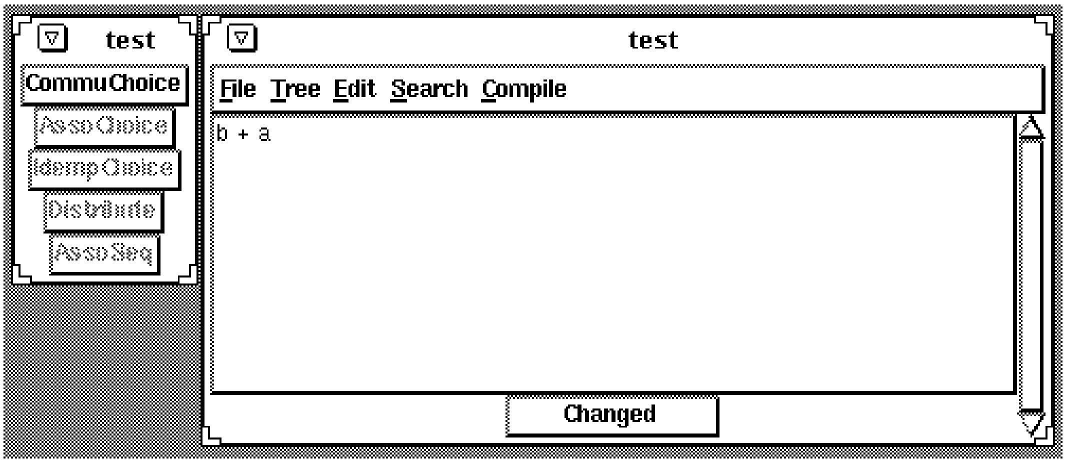

Figure 9: after pressing the "CommuChoice" button

8.3 Shortcomings

Although we would like the parser and pretty printer to be generated automatically, this ideal can not be reached yet. Because the ASF+SDF parser is a lot more powerful than parsers created by the lex+yacc combination, the resulting parser has to be "brushed up" manually, which is not always possible. Another problem lies in the universal pretty printer, which does not handle brackets at all (it does not know about them because it is language independent). This means that an expression like $(a + b).c$ is transformed into $a + b.c$ which is obviously incorrect.

9 Conclusions

9.1 Comparison of translation strategies

We have shown that the ToolBus is more powerful than SEAL. We presented (and implemented) two different algorithms translating SEAL scripts. The complexity of ToolBus scripts translated by the iterative scheme is much lower then those translated by the recursive scheme. The price paid for this decrease in complexity is a higher volume of traffic over the ToolBus when executing conditions, because every atomic condition must be send over the ToolBus three times instead of once: to the condition tool, from the condition tool to the controlling ToolBus process and from the controlling process to the tool that decides on the atomic condition. Although a typical SEAL script has a lot more actions than conditions, every change in the state of the editor requires a number of conditions to be recomputed, so this increase in ToolBus traffic may have a big impact on the overall performance (especially on the response time). Elimination of the store tool gives the iterative method a definite advantage when executing actions.

9.2 Writing ToolBus scripts by hand

It has been suggested that it might be profitable to drop SEAL and control the user interface directly from ToolBus scripts. Eliminating the SEAL translation would probably give the user a better understanding of the resulting ToolBus script, because he wrote it by hand. The price paid for this greater understanding of the resulting system would be the increased burden on the user to write ToolBus scripts directly instead of more specialized (and therefore less complex) SEAL scripts. Past experience has learned that the complexity of most "real-life" SEAL scripts is probably not high enough to warrant the translation step. Further additions to the ToolBus syntax can probably reduce the complexity of hand written ToolBus scripts to a very usable level. Possible additions include:

- An if/then/else construct. The translation of SEAL conditions is very complex. An if/then/else construct added to the ToolBus would simplify this translation considerably.

- A better data modeling capability. The store tool as introduced in Section 5.2.4 seems very inelegant, and a better data modeling capability would surely help. We might even consider some sort of type system.
- Another loop construct, maybe something a little bit closer to the perception of the programmer.
- Destruction of processes. The process controlling a start-stop button has to check after every action whether the button is still active. It might be possible to just destroy this process when the user deactivates the start-stop button.

When considering these additions, we have to be careful that they do not affect the theoretic basis of the ToolBus. Although these extensions might completely eliminate the need for SEAL, we expect that the translation constructs found in this document will be helpful when manually writing ToolBus scripts that control user interface objects.

9.3 Future Work

Future work should probably focus on manually written ToolBus scripts for controlling user interfaces, and the integration of these scripts with the other components of the ASF+SDF Meta-environment. The simple editing model (gse tool) that we adopted should probably be refined. A division into a structure editing component and a text editing component seems logical.

Appendix

A SEAL syntax in SDF

```
exports
sorts
  String Module Sort Variable-name Name
  Function-name Dir Dirs Unix-filename
lexical syntax
  "%%" L-Char* "%%"                      -> LAYOUT  %%comment
  [ \t\n]                                -> LAYOUT

  "\\" ~ []                              -> EscChar
  "\\" [01] [0-7] [0-7]                  -> EscChar
  ~ [\000-\037"\\]                       -> L-Char
  EscChar                                -> L-Char
  "\"" L-Char* "\""                      -> String

  [A-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]*     -> Module
  [A-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]*     -> Sort
  [a-zA-Z] [a-zA-Z0-9]*                  -> Variable-name
  [a-zA-Z] [a-zA-Z0-9]*                  -> Name
  [a-zA-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]*  -> Function-name

  [a-zA-Z0-9_\-]+ "/"                    -> Dir
  "../"                                  -> Dir
  "/" Dir*                               -> Dirs
  "./" Dir*                              -> Dirs
  Dir+                                   -> Dirs
  "\"" Dirs [a-zA-Z0-9_\-.]+ "\""        -> Unix-filename
  "\"" [a-zA-Z0-9_\-.]+ "\""             -> Unix-filename
sorts
  SEAL-spec SEAL-spec-part Button Menu-entry Menu Entry Active-tool
  Cond-action-pair Tool-CA-pair Docu Tool-Docu Cond-part Action
  Focus-cond-expr Pattern While-cond Focus Focus-expr Focus-action Term
context-free syntax
  "Configuration" for language Module is SEAL-spec-part+ -> SEAL-spec
  Button                                                 -> SEAL-spec-part
  Menu-entry                                             -> SEAL-spec-part
  Menu                                                   -> SEAL-spec-part
  Active-tool                                            -> SEAL-spec-part

  button Name Cond-action-pair+ Docu               -> Button
  start-stop button Name Cond-action-pair+ Docu    -> Button
  menu entry Name in Name Cond-action-pair+ Docu   -> Menu-entry
  menu Name ":" Entry+                             -> Menu
  Name "," Cond-action-pair+ Docu                  -> Entry
  active tool Name Tool-CA-pair+ Tool-Docu         -> Active-tool
```

```
when Cond-part enable {Action ";"}+              -> Cond-action-pair
enable {Action ";"}+                             -> Cond-action-pair
when Cond-part do {Action ";"}+                  -> Tool-CA-pair
do {Action ";"}+                                 -> Tool-CA-pair

Focus-cond-expr is Sort                          -> Cond-part
Focus-cond-expr matches Pattern                  -> Cond-part
String                                           -> Pattern
Focus-cond-expr "is-meta-var"                    -> Cond-part
Module ":" Function-name({Focus-cond-expr ","}*) -> Cond-part
not Cond-part                                    -> Cond-part
"(" Cond-part ")"                                -> Cond-part {bracket}
Cond-part and Cond-part                          -> Cond-part {left}
Cond-part or  Cond-part                          -> Cond-part {left}

focus {Focus-action ","}*                        -> Focus-cond-expr
Unix-filename "." focus {Focus-action ","}*      -> Focus-cond-expr

Focus-cond-expr is Sort                          -> While-cond
Focus-cond-expr matches Pattern                  -> While-cond
Focus-cond-expr "is-meta-var"                    -> While-cond
Module ":" Function-name({Variable-name ","}*)   -> While-cond
focus {Focus-action ","}+                        -> While-cond
Unix-filename "." focus {Focus-action ","}+      -> While-cond
Variable-name "." focus {Focus-action ","}+      -> While-cond
not While-cond                                   -> While-cond
"(" While-cond ")"                               -> While-cond {bracket}
While-cond and While-cond                        -> While-cond {left}
While-cond or  While-cond                        -> While-cond {left}
Focus ":=" Term                                  -> Action
Variable-name ":=" Term                          -> Action
Variable-name ":=" Focus {Focus-action ","}*     -> Action
Focus-expr                                       -> Action
while While-cond do {Action ";"}+ od             -> Action
Module ":" Function-name({Variable-name ","}*)   -> Action
create(Unix-filename, Module, Text, Sort)        -> Action
create(Variable-name, Module, Text, Sort)        -> Action
create(Unix-filename, Variable-name)             -> Action
create(Variable-name, Variable-name)             -> Action
create(Unix-filename, Term)                      -> Action
create(Variable-name, Term)                      -> Action

Focus {Focus-action ","}*                        -> Focus-expr
{Focus-action ","}+                              -> Focus-expr

focus                                            -> Focus
Unix-filename "." focus                          -> Focus
Variable-name "." focus                          -> Focus

up                                               -> Focus-action
down                                             -> Focus-action
previous                                         -> Focus-action
```

```
  next                                              -> Focus-action
  root                                              -> Focus-action
  save                                              -> Focus-action
  restore                                           -> Focus-action

  Variable-name                                     -> Term
  select "(" Unix-filename "," Sort ")"             -> Term
  select "(" Variable-name "," Sort ")"             -> Term
  Module ":" Function-name({Variable-name ","}*)    -> Term

  String+                                           -> Text
  readfile "(" Unix-filename ")"                    -> Text
  readfile "(" Variable-name ")"                    -> Text

  doc ":" String manual entry ":" Name              -> Docu
  doc ":" String                                    -> Docu
  manual entry ":" Name                             -> Tool-Docu

priorities not Cond-part -> Cond-part >
           Cond-part and Cond-part -> Cond-part >
           Cond-part or Cond-part -> Cond-part
priorities not While-cond -> While-cond >
           While-cond and While-cond -> While-cond >
           While-cond or While-cond -> While-cond
```

References

[BK94] J.A. Bergstra and P. Klint. The toolbus - a component interconnection architecture -. Technical report, University of Amsterdam, Department of Mathematics and Computer Science, Programming Research Group, 1994.

[BW90] J.X.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Kam94] J. F. Th. Kamperman. Gel, a graph exchange language. Technical report, Centrum voor Wiskunde en Informatica (CWI), 1994.

[Kli93a] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Kli93b] Paul Klint. The toolbus - a software application architecture - some preliminary thoughts -. Programming Research Group, unpublished manuscript, 1993.

[Koo92] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. Report P9222, Programming Research Group, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Koo92a.ps.Z.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[SvVvW93] P.N. Vriend S.F.M. van Vlijmen and A. van Waveren. Control and data transfer in the distributed editor of the asf+sdf meta-environment. Technical report, Programming Research Group, University of Amsterdam, 1993.