Control and Data Transfer in the Distributed Editor of the ASF+SDF Meta-environment

S.F.M. van Vlijmen
P.N. Vriend
A. van Waveren
E-mail: vlijmen@fwi.uva.nl, vlijmen@phil.ruu.nl

Abstract

For the ASF+SDF Meta-environment [K93] a new editor that combines text editing and structure editing was designed and implemented [BK93] [Koo94]. The new editor is not a monolithic piece of software, as the previous editor was, but constructed from three interacting components: a user-interface, a text editor and a structure editor. During the implementation of the editor it turned out that the interaction between the user and the editor components and between the editor components themselves was more complex than expected. In this paper an approach is presented to model the interactions between components. Following this approach, a specification of the editor, with emphasis on the most important interactions, is given in PSF [MV89, MV90, M91] and in Synchronous Interworkings [MWW92]. In conclusion, these efforts helped in finding a simple and reliable implementation of the interactions.
1991 Mathematics subject classification: 68Q40, 68Q45.
1991 CR Categories: D.1.3, D.2.1, F.1.2.
Key words & Phrases: Distributed Editor, ASF+SDF Meta-environment, Synchronous Interworking, Abstract Data Types, Process Algebra, PSF.

1 Introduction

The ASF+SDF Meta-environment is a generator for interactive programming environments [K93]. A programming environment, including the ASF+SDF Meta-environment itself, consists of a set of cooperating tools. Typical tools found in a programming environment are an editor, a parser, a type-checker and a code generator. The editor that is used in the currently distributed ASF+SDF Meta-environment is called the generic structure editor, or GSE for short [Koo92]. It is generic because it is instantiated by a syntax definition.

In [BK93] and [Koo94] it is stated that experience with GSE revealed the need for an improved version of GSE. Problems with older versions of GSE were found in maintainability, less text editing functionality than offered by commonly used editors, and the effort for a user to learn the commands of yet another text editor. A solution to these problems is presented in [BK93] and [Koo94]. The idea is to use an existing text editor to perform all text editing and displaying functionality, use a separate user-interface to handle specific interaction with the user, and use a modified version of the old GSE to perform the structure editing and the

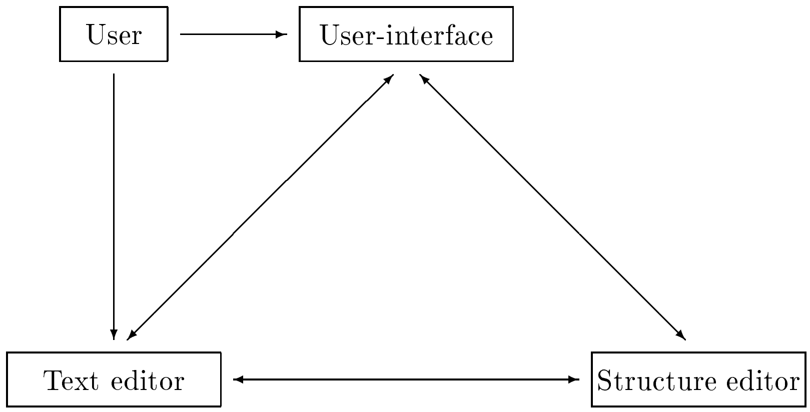


Figure 1: The architecture of GSE

control of the whole. Older versions of GSE were stand-alone and self-contained; now GSE has become a distributed program. In Figure 1 the architecture of the new GSE is depicted. As text editor Epoch [Epo92] is chosen, an extended version of Emacs.

The abbreviations 'TE', 'UI' and 'SE' will be used to refer to the text editor, the user-interface and the structure editor respectively. 'GSE' refers to the new editor as a whole. This terminology differs somewhat from the terminology of the people who developed the editor, they refer with 'GSE' also the structure editor component.

The communication behaviour between the 'participants' depicted in Figure 1 was more complex than expected: prototype versions of the editor contained deadlocks. A thorough investigation of the communication behaviour was therefore in place and forms the motivation of the work presented here. Rules that the editor components should obey in order to standardise their interaction will be given, it is investigated precisely which communications take place between editor components, and lastly the most important communication behaviour is modelled in Synchronous Interworkings [MWW92] and in PSF [MV89, MV90, M91].

We now define two notions to ease the discussion on GSE and on the specification. Although the definitions below may have to be sharpened, they may help in identifying the specific system parts and system behaviour that is investigated for the editor.

Definition 1.1. Define a system as a set of concurrently executing and cooperating components. Define a *task* of a system as the activity that is performed by a component on receiving a command from the user or from another component. With an *interaction* is meant the communication behaviour between components and between a user and components performed during the execution of a task. □

Definition 1.2. An *interaction model* is a description of the means that serve to realise interactions between the components of a system and the rules that components and the means should obey. Typical notions are the topology of components and communication channels, the protocols used in interactions and the data structures that can be communicated. □

Some simplifications on GSE are made in this paper. Typically there will exist in a system a number of instances of the editor. Some of them closely coupled, one editor containing a syntax specified in SDF, the other containing equations over this syntax in ASF [BHK89a]. Neither is this coupling studied, nor is the relation of the editor with its environment studied. The editor in this paper is considered a stand-alone editor for a single text. Syntax information used by SE is considered available. As the emphasis is on the interactions, the modelling of data types like the text, parse trees etc. are kept to a minimum, if not completely discarded.

After an analysis of the most troublesome interaction, namely the *Quit* interaction, some ideas that seemed to be useful in modelling the interactions were fixed. As 'architecture' is such an overloaded and wide notion, these ideas are called the *interaction model* (see Definition 1.2 above). A number of interactions were easily written down following the chosen interaction model. However, not all interactions fitted so nicely in the interaction model. By simulation of the specification a deadlock was detected when a Parse interaction from TE and a Quit interaction from the UI were started in parallel. This deadlock could indeed be generated in the implementation [1]. In the specification this conflicting situation is circumvented by disallowing the initiation of the Parse interaction from TE. The interactions specified are the interactions that listen to the names Quit, Expand, Parse, Window resize and Change text.

1.1 Related work

Most closely related to this work is [BKli94]. In this paper a software integration architecture is presented called the ToolBus. This is a generic means for the interconnection of components in interactive programming environments. This paper also provides an overview of work in the field of software integration, we therefore refer to [BKli94] for related work.

1.2 Paper overview

The specification techniques used in this report, Synchronous Interworkings and PSF, are introduced in Section 2. Section 3 gives an overview of GSE. Section 4 presents the interaction model. Next, in Section 5 the Quit, the Expand and the Parse interactions are discussed in general. Only for Window resize and Change text the interworking is given in Section 5. In Section 6 an overview of the specification is given. Furthermore it is discussed how specific parts of the interaction model are specified. In Section 7 conclusions are drawn and is investigated what might be interesting future research questions. In Appendix A to C the interworkings for the Quit, Parse and Expand interaction are given. In Appendix D the specification is given.

2 The specification techniques used

In this section a short overview is given of Synchronous Interworkings and of PSF. Interactions were first modelled as synchronous interworkings before they were formulated in PSF. From now on we just say 'interworking' instead of 'synchronous interworking'. The introduction

[1] The implementation was performed in parallel with the specification effort.

on PSF is a reprint. The first version of this introduction was written by W.J. Fokkink and J.J. van Wamel [FvW92], others contributed to this text [PV93] [H93].

2.1 Synchronous interworkings

To get acquainted with interworkings some of the very basics are introduced here. An interworking is often depicted with a drawing in which there are one or more named vertical lines representing processes. These processes are called entities in interworking jargon.

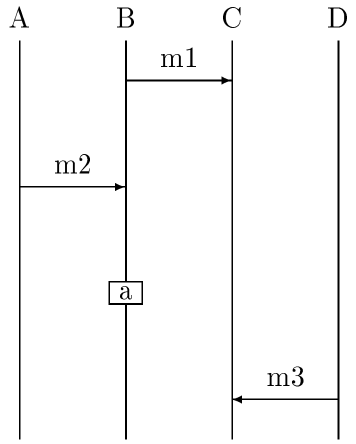


Figure 2: Example of an interworking

Two processes can communicate synchronously a message which is represented by a named horizontal arrow starting at the sender and pointing to the receiver. A non-communication action of a process is represented by an identifier located on the line that represents the process.

These notions are exemplified in Figure 2. In this figure one has processes A, B, C and D, communications of messages $m1$, $m2$, $m3$ and non-communication action a.

Operations on interworkings are the interworking sequence and interworking merge. In this paper such operations are not used and they will not be discussed here, see [MWW92].

One may wonder whether the actions in Figure 2 are executed in the strict sequential order the drawing suggests. So, how is the interworking depicted in Figure 2 to be understood? The actions in an interworking are partially ordered in time, from top to bottom. The interworking depicted in Figure 2 is semantically equal to the one in Figure 3. The idea is that actions are related if they share an entity. In the example, $m3$ is related to $m1$, but not to $m2$ and a. As a result $m3$ can freely be interleaved with $m2$ and a. This is directly visible in the object that is assigned to the drawings in the semantics as defined in [MWW92]:

$$
\begin{array}{lllllllll}
c(B,C,m1) & \cdot & (& c(A,B,m2) & \cdot & (& act(a) & \cdot & c(D,C,m3) \\
 & & & & & & + & & \\
 & & & & & & c(D,C,m3) & \cdot & act(a) &) \\
 & & & + & & & & & \\
 & & & c(D,C,m3) & \cdot & & c(A,B,m2) & \cdot & act(a) &)
\end{array}
$$

a term over Basic Process Algebra [BW90].

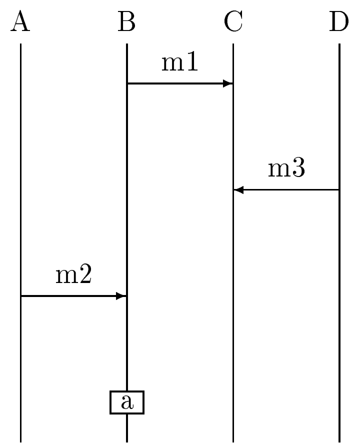


Figure 3: Another drawing for the interworking in Figure 2

2.1.1 Extending the Synchronous Interworking notation

Modelling the interactions with interworking gave some small problems. These problems stem from the modelling of choices, the absence of recursion and the relation to the specification in PSF. About the first problem, if we have that A sends $m4$ to B, *or* sends $m5$ to B, then we model this in BPA as $c(A,B,m4)+c(A,B,m5)$. However, this amounts to two interworkings.

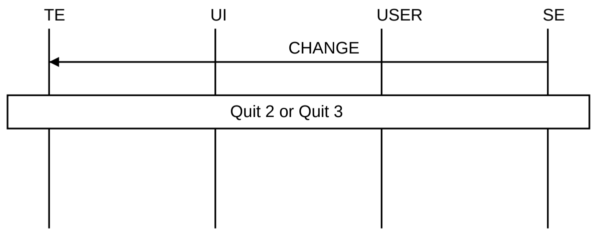


Figure 4: Use of a macro name as a shorthand to denote multiple interworkings

An informal trick to handle this, is to use the 'MACRO' facility of the interworking notation. In the name of the macro — when not expanded! — one denotes the drawing one is looking at denotes multiple interworkings. See Figure 4 for an example taken from the quit interactions in the Appendix. The drawing in Figure 4 thus stands for the two interworkings depicted in Figure 5.

The recursion problem cannot be 'solved' likewise. One could use a macro to 'do' the recursive call. However, recursion blurs the semantics of interworkings and therefore this use of a macro will not be used.

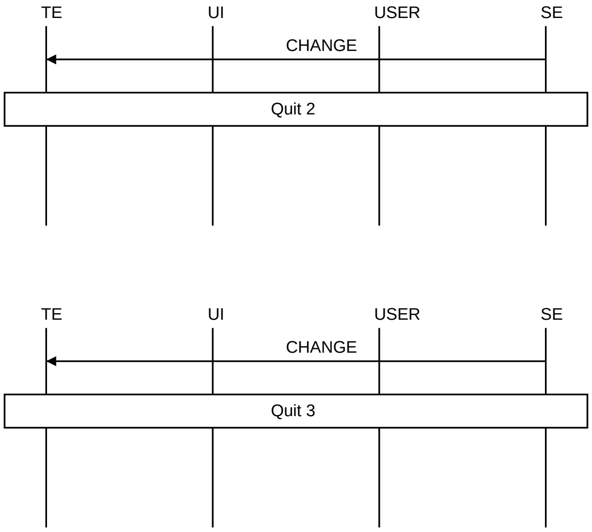


Figure 5: The interworkings denoted by the drawing in Figure 4

2.1.2 Synchronous Interworkings and PSF specifications

The relation between PSF and interworkings has been partially clarified in [VW93]. One can generate traces of an interworking in the same way as this is possible for, e.g., a BPA term. Simply take the BPA term that is associated with the interworking and generate the traces of this term. Suppose we have a specification that obeys certain criteria, e.g., no data specification is part of the specification. It is shown in [VW93] that a finite trace of this specification can be translated to an interworking, and that the traces of this interworking can all be generated by the specification. Because the PSF specification in this paper does specify data types and because data objects are used as argument to atomic actions, this fact is not applicable. The only thing that can be said is that we do not expect that the interworkings in this paper specify traces that cannot be generated by the PSF specification. It is more likely that the PSF specification specifies traces that cannot be generated by the interworkings. In case of doubt, the PSF specification has the last word. It was checked by hand that the order in which actions appear in the interworkings, reading from top to bottom and left to right, form traces of the PSF specification of the editor.

2.2 The specification language PSF

The specification language PSF, a Process Specification Formalism, has been developed at the University of Amsterdam. It is a combination of an algebraic specification formalism for data, and process algebra. Therefore, PSF has an explicit data-part and a process-part using the data. Some general references to PSF are [MV89], [MV90], [MV93] and [Vel93].

2.2.1 Data in PSF

This section contains a brief introduction to the specification of data by algebraic specifications, as is prescribed in PSF. More detailed background information can be found in [BHK89a] and [BHK89b].

Data are time-independent objects with mathematical properties such as symmetries, invariants and representation results. In PSF, data are constructed from a given, finite number of *constants* and *functions*. Furthermore, each data term is typed by a sort name (such as Naturals or Booleans). Properties of data can be defined by *equations* of the form $d = d'$, where d and d' are data terms of the same sort that may contain (typed) variables.

For example, let Nat be the sort of the naturals, with a constant 0 of type Nat and a function $S : \text{Nat} \to \text{Nat}$, denoting the successor function. No equations are specified. Then the *initial algebra* (representation of all closed terms modulo provable equality) of Nat is the collection

$$\{\ 0,\ S(0),\ S(S(0)),\ ...\ \}.$$

One can add a function such as $+ : \text{Nat} \times \text{Nat} \to \text{Nat}$, denoting addition, with equations

$$\begin{aligned} &n + 0 = n \\ &n + S(m) = S(n + m) \end{aligned}$$

where m, n are variables of the type Nat. In this case the initial algebra is

$$\begin{array}{ll} \{ & [\ 0,\ 0+0,\ (0+0)+0,\ ...\ ] \\ & [\ S(0),\ 0+S(0),\ S(0)+0,\ ...\ ] \\ & [\ S(S(0)),\ S(0)+S(0),\ 0+S(S(0)),\ ...\ ] \\ & \cdots \hfill \} \end{array}$$

where the collections between square brackets are the equivalence classes of the sort Nat, generated by provable equality (a congruence relation).

In general, it can be hard to decide whether or not two closed data terms are equal. A means for doing this is offered by the theory of *term rewriting systems*. One can try to set up the equations for the data in such a way that, when applied from left to right only, each reduction of a closed data term is finite. This means that every reduction ends with a term, called a *normal form*, to which none of the (left-hand sides of the) equations matches anymore. The aim is to have *unique* normal forms, i.e., if two closed data terms are equal, their normal forms are literally the same. Then it clearly is decidable for each pair of closed data terms whether or not they are equal, by simply reducing them to normal forms.

In the example of the natural numbers above, the normal forms are the expressions of the form $S(S(...S(0))...)$. Thus the closed data terms of the sort Nat have unique normal forms. However, if an equality $m + n = n + m$ is added to the equations for the $+$, one gets infinite reductions such as $0 + S(0) = S(0) + 0 = 0 + S(0) = ...$.

Modules enable the explicit declaration of algebraic data types as described above. A module consists of two parts: one in which the *algebraic signature* is given and a second which contains the *equational logic*. So a data module is a structure of the following form:

```
data module Naturals
begin

  < algebraic signature >

  < equational logic >

end
```

Signatures are sets of declarations of *sorts* and *functions*. A *constant* is simply a function with no arguments. For example, the algebraic signature of the module Naturals is as follows:

```
sorts
  NAT

functions
  0   :             -> NAT
  S   : NAT         -> NAT
  _+_ : NAT # NAT -> NAT
```

In the second part, the equations for the data terms are given. In the case of the Naturals this is:

```
variables
  m,n :      -> NAT

equations
  [N1]  n+0 = n
  [N2]  n+S(m) = S(n+m)
```

An important feature is *import*. A module can import other modules, whence an initial algebra that has been created in a first module can be used in a second one.

2.2.2 Processes in PSF

In this section a short description of process algebra is given. Process algebra forms, together with module algebra and term rewriting techniques, the algebraic basis of PSF. After a short description of process algebra, some characteristics of PSF are mentioned, and some of the tools that are available for the manipulation of specifications in PSF.
A process can be described informally as *the behaviour of a system* over time. It turns out, however, to be an intricate philosophical problem to describe the difference between processes and data in a formal way. This is especially true when processes are parameterised with data, and complex interactions between processes and data take place. In [Pon92] attention is paid to process algebras with data.
Process algebra is a formal approach to the specification and verification of concurrent systems. It may be considered the study of concurrent communicating processes in an algebraic framework. Several process algebras have been developed in the past decade, but here we

discuss only ACP, the Algebra of Communicating Processes. There are different variants of ACP. For instance, abstract or untimed ACP [BW90] and Real Time Process Algebra [BB91]. We restrict ourselves to abstract ACP, which forms part of the core of PSF. Other approaches to process algebra can be found in [Hoa85], [Hen88] and [Mil89].

We briefly introduce ACP. The theory of ACP has two parameters: a set of *actions* A, and a partial communication function $\gamma : A \times A \rightarrow A$ that is commutative and associative. Actions are the most basic *processes* considered, and are not assumed to be subject to further division. Three important binary operators to compose processes into more complex ones are sequential composition ($\cdot$), alternative composition or choice ($+$), and parallel composition ($\|$). Particularly, $p \parallel q$ represents the process that either performs a first action of p, or of q, or a 'communication action' of p and q, i.e., the simultaneous occurrence of a first action of p and one of q. There are also special constants in process algebra, the most important of which are the *deadlock* (δ) and the *silent step* (τ). If a system can perform no action but a deadlock, it is blocked: it cannot continue or terminate successfully. The silent step denotes an *internal* action, i.e., an action that is hidden (not observable). Two system operators must also be mentioned here: *encapsulation* and *hiding*. Encapsulation forces the components of a parallel system to communicate and thus to synchronise. Hiding renames all internal actions of a system into a silent step. Once the signature of a process algebra is defined, an axiom system can be defined. An axiomatic *verification* of a system is a formal proof of the fact that the observable behaviour of the system (abstracting from internal actions) satisfies a given requirement. Algebraic verification can be performed simply by manipulating equations, using the axioms of a given algebra. Another option is verification in some specific *model* for processes (a process algebra). Well-known models for processes are graphs or labelled transition systems (after dividing out some behavioural congruence such as *bisimulation*). Practical examples of specifications and verifications of concurrent systems can be found in the literature. Examples of such systems are protocols in computer networks, systolic algorithms, and logistic protocols in factories. Some references are [Bae90] and [W92].

The formal specification language PSF is based on abstract ACP. It was developed in order to provide ACP with the means to write tools for the verification and simulation of specifications. The formal definition of PSF also makes it possible to verify whether a specification is syntactically correct or not. The definition of PSF can be found in [MV90].

A PSF specification is organised in *data modules* and *process modules*. The different modules of a specification are connected via an import-export mechanism. The *exports* section of a module contains the visible part of the algebraic signature of a module when it is imported into another module via its *imports* section.

In process modules, the processes of a specification are specified. For example: if a system SYS consisting of a sender S and a receiver R is specified, S can be put in one process module and R in another. Then the process SYS can be specified in a *top-module*, which means that it forms the highest level in the specification in which all other modules are imported.

In data modules, the abstract data types used in a process specification are defined. No standard types (*sorts* in PSF) such as naturals or reals are available. This part of PSF is not based on process algebra but on module algebra and term rewriting techniques, as described in the previous section. In process modules, data may be imported and used in various manners to specify processes: communication can be defined in data parametric way; processes can be defined recursively in a data parametric way with tests on data values; and there is a 'sum' operator for defining the alternative composition of the closed data-instantiations of a parametric process.

The specification language PSF is equipped with several tools for the analysis of a specification. We mention some of these that are typical for analysis by simulation:

1. The *term rewriter*. By means of this tool, expressions are rewritten to normal forms, which is useful to verify whether or not functions have been defined correctly.

2. The *initial*-tool, which makes it possible to calculate (part of) the initial algebra of a specification. It calculates the different elements of a given sort (type), using the functions and the rewrite rules defined on these functions.

3. The *simulator*. This tool makes possible a stepwise analysis of a specification. The simulator performs successive actions that the user can choose from a menu. To put it more technically: with the simulator traces through the *state-space* of a system can be generated. The simulator can also make random runs. This option yields, in general, a fast method to detect deadlocks or other undesired system behaviour.

Tools 1 and 2 are suitable for the analysis of the data in specifications.
Examples of PSF specifications can be found in e.g. [Bru93], an other source is the report series of the Programming Research Group of the University of Amsterdam. For more detailed information on the syntax of PSF the reader is referred to [MV90] and [MV93].

Finally, it must be remarked that no real time features are captured. Consequently, important features like *efficiency* (maximal productivity of machines) and *tuning* (synchronisation of the speed of the machines) cannot be modelled. Currently, the essential realtime features have been added to process algebra [BB91]. Real space features have been added to process algebra as well [BB93].

3 An introduction to GSE

First some aspects of TE, UI and SE are highlighted that are important for understanding the specification and the interworkings in the appendices. Second the most important interactions are introduced.

3.1 The user-interface

UI serves to present a uniform interface to the editor components TE and SE. Here we focus on the role of the user-interface, UI, in the editor. UI offers mainly three things: windows to display text, decoration, e.g., buttons, scrollbars, to these windows to access functions in SE, e.g., Quit, Expand, Parse, and dialog boxes for various purposes, e.g., file browsers. Not all communication with the user is via UI. Text editing by TE, as opposed to structure editing, like inserting a character, is directly performed by TE. In other manipulations of window contents, like clicks on buttons, window resizing and use of scrollbars, UI does play a role. However, UI has not much knowledge about the meaning of these manipulations and therefore informs TE or SE about their occurrence. E.g., when the window is resized TE is informed about this, and TE will in response redisplay its contents.

3.2 The text editor

Texts that are manipulated by the editor are located in the text editor, TE. TE also performs the actual manipulations on the text in response to commands issued by the user or by SE. Furthermore, TE displays, loads, and stores texts. To realise structure editing two data structures are used for each text: a *parse tree* and a *zone list*. The parse tree is kept by SE, the zone list by TE.

$$\underbrace{\ldots\ \underbrace{if\ \underbrace{x\ =\ \underbrace{\boxed{1}00}_{zone\ n}}_{zone\ n-1}\ then\ A\ else\ B}_{zone\ n-2}\ \ldots}_{zone\ n-3}$$

Figure 6: Example of zones and cursor position

Given a syntax-correct text and its corresponding parse tree. Suppose it is known for every subtree of the parse tree what part of the text has lead to the subtree in the parse process. Call this information the *zone* of the subtree. Given a certain cursor position in a syntax-correct text in TE, the zone list contains all the zones in which the cursor position is contained. In Figure 6 the expression '$if\ x\ =\ 100\ then\ A\ else\ B$' is displayed in some context '... ...'. The zones are calculated during the parse process. The zone list for a specific cursor position is communicated from SE to TE after a successful parse. A successful parse brings TE in the *struct state*. There are two other states *text state* and *error state*. When TE is in struct, text or error state, the editor as a whole, GSE, is considered to be in struct, text or error state respectively.

There is one zone of particular importance, the *focus*. The focus is the smallest zone enclosing the cursor position. The focus text is displayed in a colour that differs from the colour of the text surrounding the focus. The colour chosen is dependent on the state (error, struct, text).

In struct state, just after a parse, the focus text is precisely the text corresponding to the smallest subtree in the parse tree containing the cursor. In Figure 6, the focus is zone n. The focus is not fixed, another zone may become the focus by text editing, structure editing, navigation or parsing. In struct state navigation is stepping through the parse tree; zone list and focus get updated accordingly. In error or text state navigation is just cursor movement inside the focus, and focus enlargement when one tries to move the cursor outside the focus. So, the focus will contain always unparsed text when navigating in error or text state, only in struct state one can freely position the focus.

During text editing the focus may change when its boundaries are crossed. When this happens the focus is *enlarged* to the smallest enclosing zone. To illustrate this, consider what happens in the situation depicted in Figure 6 when the cursor is positioned one character to

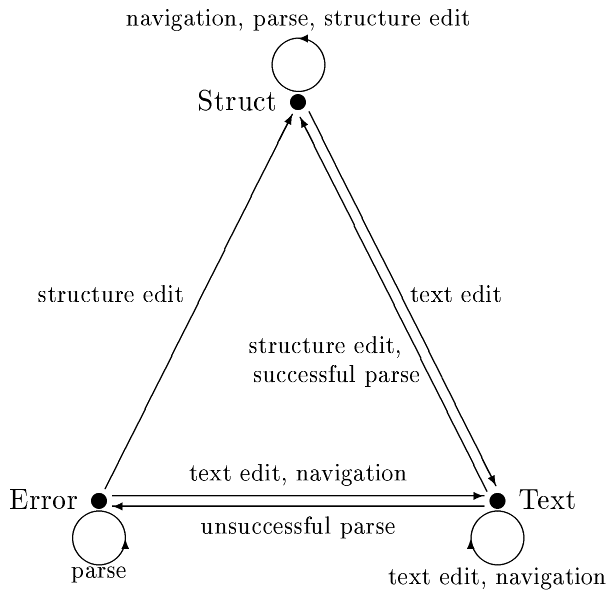


Figure 7: relation of editor states and main commands

the left or when the first character to the left is deleted. The new focus will become zone $n-1$. Herewith it is always known which part of the text is correct, and so what part is only to be parsed.

Structure editing is possible in text, error and struct state. The relation between the states and the commands navigate, parse, structure edit, text edit are depicted in Figure 7.

During structure editing the focus can change when the focus text is replaced by a template of an expression of the same type as the focus text (over syntax that is available to the editor). This means that the contents and boundaries of the focus and enclosing zones change. The position of the focus in the parse tree does not change. See the discussion on the expand command in Section 3.3.

3.3 The structure editor

The structure editor offers, as one might expect, structure editing, but is also controlling TE en UI. Furthermore, SE forms the connection to other components in the ASF+SDF Meta-environment or generated environment. These connections are however not modelled in this paper. There is no direct communication between a user and SE, all dialogs run via UI or TE.

A typical control activity of SE is found in the role of SE when the editor is asked to terminate. The communication that is performed during the termination process is modelled in the Quit interaction. Another control activity, not modelled in this paper is, e.g., the instantiation of an editor upon request of the ASF+SDF Meta-environment.

As GSE offers text editing, the text may not always be syntax correct. In text mode SE can be instructed explicitly to parse the (focus) text. The communication that is involved is modelled in the Parse interaction. In error or struct state a parse just leaves the editor in error or struct state respectively.

When the editor is in struct, error or text state, one can navigate and perform structure editing. Navigation is not modelled in the specification, structure editing is, to some extend that is. Structure editing is performed by the Expand command. All communication that takes place during the execution of the expand command is modelled in the Expand interaction. Suppose the focus text is or is expected to be of type T. In this situation SE responds to an expand command by offering a palette of templates of possible expressions over the syntax of type T. This palette is displayed by UI, and the user is asked to select a template. After this, the text in the focus will be replaced by the template chosen.

4 An interaction model for the editor

First, an interaction model is described in an abstract way. The interaction model fits the intuition one has with the well-known — not necessarily well-defined — notion of client-server architecture. Second, the editor specific instantiation is discussed. This Section ends with an example taken from the Quit interaction.

In the interaction model components are connected by bidirectional communication channels. There is no restriction on the topology of the connection pattern. Communication on these channels is performed in pairs. The first communication is called an *event*, the second communication is called a *response*. Components can also receive events from a user but these events are not followed by a response. Events and responses are not broadcasted, but are always sent on a named channel to a known receiver. After a component A has issued an event to a component B, component A waits for the response from B. Component B on its turn receives an event, B is then allowed to perform internal actions. As a result B is not allowed to set up a (nested) communication pair with a third component before responding to A. When events and responses are the only visible actions, an interaction is a sequence of communication pairs interspersed with events from a user.

The state of a component is for a part determined by the set of events the component is willing to accept. This set of events is called the *event state* of the component. In the interaction model it is allowed that components change their own event state, but they may also change the event state of other components. This offers the possibility to let components control a part of the behaviour of other components. This control is explicitly modelled, as will be explained now. Suppose component A wants to change the event state of component B, then A sends operations to B that instruct B how to change its event state. With these explicit event state operations, it can be circumvented, at least in a specification, that B changes its own event state on the basis of semantical information of A that B is not entitled to know.

By definition a component should always be willing to receive events that specify a modification of its event state. In other words, the event state should always contain these events.

One can distinguish between four types of communication pairs. The difference is in the message type communicated by the event and by the response. Distinction is made between a *change of event state*, a *notification*, a *data item* and an *acknowledgement*. In Figure 8 the four meaningful combinations of event and response are listed.

Interactions are sequence of communications pairs interspersed with events from a user.

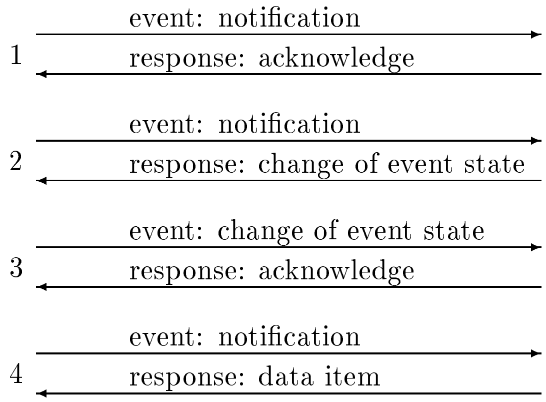


Figure 8: The four kinds of communication pairs.

This seems quite restrictive, and maybe it is. The reason to impose this restriction is the unproven idea that this may lead to simple communication behaviour. Some forms of nesting can, however, still be achieved. Suppose A sends an event to B. When B needs to consult C before it can respond to A, then B explicitly puts A in a state in which A is not harmful. That is, a state in which A cannot disturb the activity of B. The extreme here is that B puts A in a state where A is only able to receive events from B. Modelling in this way takes six communications (A to B, B to A, B to C, C to B, B to A, A to B) instead of four in the case of nesting (A to B, B to C, C to B, B to A). From an implementation point of view this may seem a waste of time. From a specification point of view, the interaction model forces, in our view, the explicit specification of an important form of control between components. Furthermore the model is still flexible, e.g., in the example B could allow A to perform some activity provided this activity is not in conflict with activity of B or C.

The interaction model has now been introduced in general. How do we use the model in the specific situation of the editor?

The connection pattern of the components is already presented in Figure 1. Note that the user does directly issue events to TE but not to SE. SE is however in control of the system, and in this position SE will change the event state of UI, TE and of itself from time to time. A part of the Quit interaction is given in order to exemplify the ideas presented in this Section. In Section 6 the specific abstract data types and processes of the specification of the editor are discussed. A discussion on the Quit interaction is found in Section 5.1. The complete interworkings of the Quit interaction are given in Appendix A.

Example 4.1. Suppose that the editor has just started executing when the Quit interaction begins; the `QUIT` command is the first command that is sent by the user. In that case the components of the editor are in their initial event states. These event states are the following, note that only the events that are of importance to the example are listed:

- {`USER^QUIT, USER^EXPAND, USER^PARSE, USER^WINDOW-RESIZE, ...` }, for UI;
- {`UI^QUIT, UI^EXPAND, UI^PARSE` }, for SE;
- {`USER^CHANGE-TEXT, SE^SAVE-FILE, ...`}, for TE.

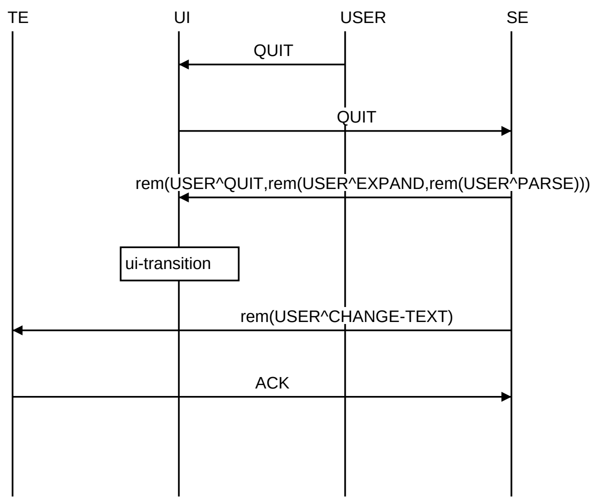


Figure 9: A number of actions from the start of the Quit interaction

In Figure 9 the interaction starts as the user sends a `QUIT` command to UI. Because `USER^QUIT` is in the event state of UI, UI accepts the command, but UI is not in control of the editor, thus UI retransmits this command to SE. As `UI^QUIT` is in the event state of SE, SE accepts this notification, as this event is called according to the terminology in Figure 8. This transmission is the first communication of a communication pair, so UI is waiting for SE to respond. The response of SE to UI is a command to change the event state of UI. The response tells UI not to accept a `QUIT`, `EXPAND` or `PARSE` command from the USER; `rem(USER^QUIT,rem(USER^EXPAND,rem(USER^PARSE)))` should be read as 'remove the events `USER^QUIT`, `USER^EXPAND` and `USER^PARSE` from the current event state'. The change of event state by UI is modelled with the action `ui-transition`. After this change, the event state of UI is {`USER^WINDOW-RESIZE, ...`}. The next communication pair is initiated by SE: SE forbids TE to allow further editing on the text, TE just acknowledges the received event from SE. Note that `ui-transition` occurs in parallel with `rem(USER^CHANGE-TEXT)` and with `ACK`, and not strictly before this communication pair, as the interworking interpreted as just a drawing might suggest (see Section 2.1). □

5 An introduction to the modelled interactions

In this Section the interactions modelled in PSF and in interworkings are introduced in general. The presentation is not detailed about the specific communications that take place in a specific interaction. This detail could have been easily added by interspersing the text with the interworkings of the specific interaction. We have however chosen to give the interworkings in the appendices, because we expect that the detailed information on the interactions is best presented by the interworkings when they are presented closely together, as in the appendices. Only the extremely simple interworkings of the Window resize and Change text interaction

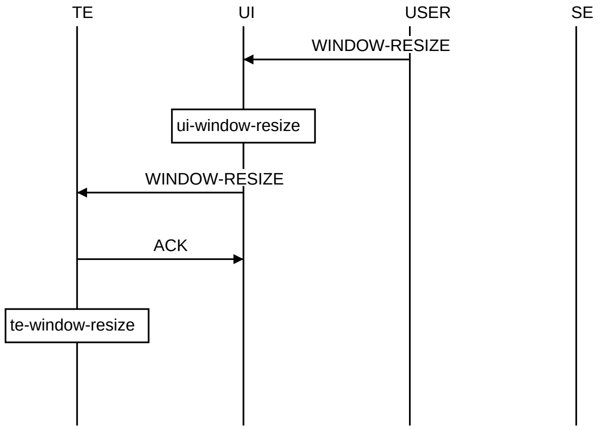


Figure 10: The window resize interaction

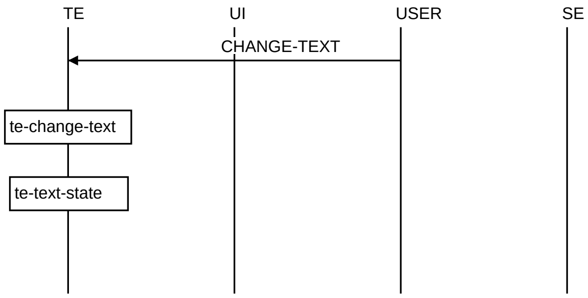


Figure 11: The change text interaction

are displayed in this Section, more or less to give a taste of the modelling of interactions by interworkings. Furthermore, some differences with the interactions found in the 'real' editor are discussed as well as interactions that are not modelled.

To start with the last category, there are a number of operations on text and windows that do not change the text. There is only one interaction of this type modelled: the Window-resize interaction. There are a lot of text editing facilities to change text. However, they follow mostly the same communication pattern. This type of operations is modelled by the Change Text interaction.

Both the Window-resize and Change Text interactions are quite simple. Their interworkings are displayed in Figure 10 and 11 respectively. The figures are expected to be self-explanatory. We will now focus on the Quit, Parse and Expand interactions.

5.1 The Quit interaction

The Quit interaction models the communication and relevant internal steps that follow when the user has instructed the editor to terminate. The user directs the editor to terminate by clicking on the Quit button that is found on the border of the window displaying the text. Considering again Figure 9, we see that after receiving the QUIT command from UI, SE starts with disabling user-interaction with UI except for Window-resize. This kind of actions are not disabled because the user may want to look at the text one more time before deciding to save a file. The next step is to disable text changes directed to TE. If the text in the buffer of TE is not changed since the text was loaded, then SE instructs TE and UI to terminate and next SE will terminate itself. If however the text in the buffer of TE is changed a dialog with the USER is started. Typical dialogs try to answer questions like whether the user wants to save text, if so, whether the user really wants to overwrite a file of the same name, etc..

5.2 The Parse interaction

The user directs SE to parse the focus text by clicking on the Parse button. This button is positioned just like the Quit button on the border of the window displaying the text. In the 'real' editor a parse can also be activated from TE by typing 'ESC P' or double-click. Use of one of these commands in combination with a click on the Quit button of UI may lead to a deadlock. 'ESC P' or double-click has been left out in the specification. In Section 7 is elaborated on this deadlock.

Just as in the Quit interaction, SE is the component that takes the lead. After SE has received the request to start parsing, SE will disable events to TE and UI that can interfere with the parse process, next SE will ask TE in what state it is. When TE is in error or struct state SE will not execute a parse, as this would obviously not lead to any new insights, see Figure 7. When TE is in text state, a parse is started. To that end, the focus text is sent from TE to SE and parsing is started. If during parsing an ambiguity is detected, a dialog with the user is started. A set of the possible ways to continue the parsing is composed by SE, sent to UI, and displayed by UI in a dialog box. Also a cancel option is offered in this dialog box. The user can pick one of the options or cancel the parse. If the parse is not cancelled, then the parse process will continue. The parse process terminates when the text is successfully parsed, when an error is found or when the user selects cancel in a 'disambigue' dialog box.

If the parse was successful, then TE is instructed to move to struct state. A new zone list and a new focus are sent to TE by SE.

If the parse was unsuccessful, then TE is instructed to move to error state. No new zone list is sent to TE. The focus may however have to be enlarged, this is done by taking as the new focus one of the enclosing zones. Focus enlargement may happen when the editing in the focus changed the type of the text in one or more enclosing zones. The cursor is positioned in the focus on a position where the parser signals the faulty text. In the specification, cursor positioning and possible focus enlargement is simplified, the message `MOVE-TO-ERROR-POS` models all this, see, e.g., interworking Parse 7 in appendix B.

5.3 The Expand interaction

Expand is the structure editing facility offered by GSE. Expand is activated by clicking the Expand button of UI. Expand can be performed in text, error and struct state and will always bring the editor in struct state if the expand operation is not cancelled, see again Figure 7 above. SE starts with disabling events directed to TE and UI that can interfere with expand: the commands `CHANGE-TEXT`, `EXPAND`, `QUIT`, `PARSE` are disabled. Next, SE asks TE for information on the current focus. If SE has received this information, it is able to find out the type of the text that the focus should contain. Given this type, SE is able to construct a palette of templates of this type over the syntax. SE sends this palette to UI and after that it is displayed by UI in a dialog box. A special option that is always offered is the cancel option. Selection of this option will cancel the expand process, and nothing will be changed in the text. However, when the user selects one of the templates, the focus text is replaced with the template. Focus boundaries are changed accordingly. In the specification and in the interworkings always the same choices are offered: `CHOICE1` and `CHOICE2`. This is of course a serious simplification, but adequate because the activity is only of importance in this paper, not the precise contents of a dialog box.

6 Overview of the specification

In this Section is discussed how the interaction model from Section 4 and the instantiation of the interaction model for GSE are modelled in PSF. See appendix D.

6.1 Data modules

The specification is focussed on interaction between editor components. So, data types that are important for the interactions are modelled, other important data types like texts and syntax trees are not.

Of course there are the typical sorts one always needs: `BOOLEAN`, `NATURAL`, `PAIR` and `SET`. The first two modules are imported from the PSF-library [W93].

The editor specific data types are modelled in modules `Buffer-state`, `Participant`, `Message`, `Event` and `State-Message`. Module `Buffer-State` specifies the three states TE can be in: `text-state`, `struct-state`, `error-state`.

Sort `PARTICIPANT` from module `Participants` contains the names of the senders and receivers that figure in the specification: `TE`, `SE`, `UI` and `USER`. The objects in `PARTICIPANT` serve to identify the sender and receiver of messages.

In Section 4 the notions of event, response and event state were introduced. Also four 'types' of messages, carried by an event or by a response, were distinguished: change of event state, notification, data item, acknowledgement. The latter three 'types' are modelled as objects of sort `MESSAGE` (see module `Message`). This is a simplification, for example, a question like `CHANGE` meaning "is the text changed?", a command like `QUIT`, an answer like `OK` and a data object `TEXT` are all modelled as object constants of sort `MESSAGE`. The simplification is not harmful, e.g., we do not bother about the exact text communicated, so `TEXT` suffices. However, the former 'type', *change of event state*, is a separate sort in the specification, namely STATE-MESSAGE. The definition of this sort uses sort EVENT and sort EVENT-SET. We will now Discuss EVENT and EVENT-SET, then we return to STATE-MESSAGE.

An element of sort EVENT is a pair of a PARTICIPANT, e.g., UI, TE .., and and a MESSAGE. An element of EVENT-SET is a finite set of EVENTs. See modules Event and Event-Set. The event state of a component is an element of sort EVENT-SET. If USER^WINDOW-RESIZE is in the event state of an UI, then UI is able to receive a message WINDOW-RESIZE from USER. However, a component should always be able to receive some special messages, namely messages of sort STATE-MESSAGE. Elements from STATE-MESSAGE specify operations on elements of sort EVENT-STATE. Because a component should always be able to receive messages of sort STATE-MESSAGE regardless on who is the sender, pairs P^SM where P is in PARTICIPANT and SM is in STATE-MESSAGE are not included in the event state. Below is explained how operations on EVENT-SET are specified by elements in sort STATE-MESSAGE.

Elements in STATE-MESSAGE specify a sequence of set operations on event states, e.g., union, difference, intersection. Such a sequence of length n, $n \geq 1$, has the following form.

$$operation_1(argument_1, operation_2(argument_2, ..operation_n(argument_n)..))$$

Here $argument_i$ $(1 \leq i \leq n)$ a closed term of sort EVENT or EVENT-SET. In the former case $operation_i$ is rem, remove element $argument_i$, or add, add element $argument_i$. In the latter case $operation_i$ is union, or diff, or isect. These operations stand for the set operations union, difference and intersection respectively. Given an object of STATE-MESSAGE of the form in the above and given an event state E. Application comes down to the following.

$$operation_n(argument_n, operation_{n-1}, (argument_{n-1}, ..operation_1(argument_1, E)..))$$

In the editor, SE sends this kind of messages, but does not receive them. UI and TE receive these messages and do not send them. There is no special reason for this. In a specification that models interaction with the Meta-environment, SE will probably receive messages of sort STATE-MESSAGE.

Example 6.1. In this example is elaborated on example 4.1. Notions introduced in Example 4.1 are presented here again but now in the setting of the PSF specification.
UI, USER etc.. are terms (constants) of sort PARTICIPANT. QUIT, EXPAND and SAVE-FILE are terms (constants) of sort MESSAGE. USER^QUIT, USER^EXPAND etc.. are terms of sort EVENT. The event state for component SE in PSF is:

```
add(UI^QUIT,add(UI^EXPAND,add(UI^PARSE,empty-event-set))).
```

The message of the third communication in the interworking of Figure 9 in Section 4 is:

```
rem(USER^QUIT,rem(USER^EXPAND,rem(USER^PARSE))).
```

This communication is the response of SE to the event it just received from UI. The response specifies an operation to be performed by UI on its event state. The term

```
rem(USER^QUIT,rem(USER^EXPAND,rem(USER^PARSE)))
```

is a term of sort `STATE-MESSAGE`. Such a term specifies an operation to be applied to an event state, in this case the event state of `UI`. Let's apply the operation in the above to the event state of `UI`, which is limited to four elements for simplicity:

```
apply( rem(USER^QUIT,rem(USER^EXPAND,rem(USER^PARSE))), add(USER^QUIT,
   add(USER^EXPAND,add(USER^PARSE,add(USER^WINDOW-RESIZE,empty-event-set)))))
= by axiom ap10 of module State-Message
apply( rem(USER^EXPAND,rem(USER^PARSE)), rem(USER^QUIT,add(USER^QUIT,
   add(USER^EXPAND,add(USER^PARSE,add(USER^WINDOW-RESIZE,empty-event-set))))))
= by the axioms of module Set
apply( rem(USER^EXPAND,rem(USER^PARSE)),
   add(USER^EXPAND,add(USER^PARSE,add(USER^WINDOW-RESIZE,empty-event-set))))
= again by ap10
apply( rem(USER^PARSE), rem(USER^EXPAND,
   add(USER^EXPAND,add(USER^PARSE,add(USER^WINDOW-RESIZE,empty-event-set)))))
= etc... =
add(USER^WINDOW-RESIZE,empty-event-set).
```

□

6.2 Process modules

There are four process modules. Three of them specify the editor components UI, TE and SE. See modules `User-Interface`, `Text-Editor` and `Structure-Editor`. These three modules are imported by the fourth module, namely `Generic-Structure-Editor`. In this module the communication function and the abstraction set `H` are specified. Furthermore the parallel composition of the processes `UI`, `TE` and `SE` is specified here:

```
GSE = encaps(H, UI || TE || SE).
```

The action alphabets of the processes consist, on the one hand, of actions that may join in a communication, or that are the result of communication, and, on the other hand, of actions that model some internal activity of an editor component.

Typical actions that join in a communication are, for some `m` in `MESSAGE`, e.g.,

`event-in-TE(UI, m)` and `event-out-UI(TE, m)`.

Synchronisation of these actions results in the action `event(UI, TE, m)`. This action comes with the following intuition: `UI` initiates the event, the event is received by `TE`, and the message transported is `m`. See the definition of the communication function in module `Generic-Structure-Editor` for a convenient overview of all these actions. Note that not all action of the form `event-in-..` and `event-out-..` have to synchronise: the actions `event-in-UI(USER,m)` and `event-in-TE(USER,m)` are not encapsulated. This is because there is no process modelling a user specified that could synchronise with these actions.

Typical actions that model some internal activity of an editor component are, e.g., `ui-window-resize`, `te-terminate` or `se-update-focus-pointer`. These actions are mostly not of significance for modelling interaction of editor components. They are added to enhance readability of the specification.

$$
\begin{array}{lcl}
P(E) & = & \sum_{e \in E} read(e) \cdot Handle(e) \\
Handle(e_1) & = & \tau \cdot P(E_1) \\
. & & \\
. & & \\
Handle(e_n) & = & \tau \cdot P(E_n)
\end{array}
$$

Figure 12: Structure of an editor component

The processes UI, TE and SE are specified along the same lines, displayed below in an abstract form for some editor component P.

So, a component is able to read an event e in E. This event is handed over to a handler process. This handler process does some internal processing modelled by τ. Finally P is called recursively, with an event state that may differ from E.

The specifications of UI, TE and SE refine the scheme in Figure 12. UI and TE discriminate between reading an event with a message of sort MESSAGE and an event with a message of sort STATE-MESSAGE. The acceptable events of the former kind, are explicit in the event state. The event of the latter kind can always be read. In other words, there is no check on their acceptability. Below the first part of the specification of process UI is displayed.

```
UI(E)
= sum( p in PARTICIPANT-f,
       sum(m in MESSAGE-f, [is-in(p^m, E) = true]
           -> event-in-UI(p,m) . HANDLE(p,m,E))
       +
       sum(m in STATE-MESSAGE-f, event-in-UI(p,m) . HANDLE(p,m,E))
     )
```

Below two handlers are displayed. The first is taken from UI. The seconds is taken from TE.

```
HANDLE(SE,ADD-EXPAND-MENU,E)
= response-out-UI(SE,ACK) . ui-add-expand-menu . UI(E)
```

and

```
HANDLE(UI,WINDOW-RESIZE,E,tc,es)
= response-out-TE(UI,ACK) . te-window-resize . TE(E,tc,es).
```

The specification of SE differs from the specification of UI and TE in that SE start with initialising the event state of UI and TE, and more important, SE is not able to receive events which carry a STATE-MESSAGE. These two differences are related to one of the function SE has: the control of UI and TE.

The first example in the above is taken from module User-Interface. It is the handler for the message ADD-EXPAND-MENU from SE to UI. The second example in the above is taken from module Text-Editor. It is the handler for the message WINDOW-RESIZE from UI to TE. The parameter tc is either text-changed or text-not-changed, the parameter es is either text-state, or error-state or struct-state.

`SE` has more involved subprocesses because it controls `UI` and `TE`. Note however that there is no nesting of communication pairs. After each `event(X,Y,M)` action follows a corresponding action `response(Y,X,M')`.

In the text of module `Structure-Editor` comments are used to relate specification text to the interworkings of appendices A, B and C, e.g., `-- Quit 1` refers to interworking 'Quit 1' in appendix A. The match is not perfect, e.g., only the text of module `Structure-Editor` is related to the interworkings. However we hope this match, shaky as it is, will help the reader to quickly understand the PSF specification after he has studied the interworkings. Therefore we will also refrain from extra comments on the PSF specification.

6.3 Relation of specification and implementation

In this section the relation of the specification and implementation is addressed. If differences seem strange to the reader, he or she should remember that the specification effort was undertaken only when the implementation work was already going on for some time, and this effort was performed to analyse some problems of the editor, not to model the editor completely. The emphasis in this paper is on the Quit, Parse and Expand interaction, see section 1. Modelling of all other important features of the editor, e.g. data structures, edit operations etc., is kept to a minimum. Furthermore, communication with components outside the editor is not modelled, and initiation of a parse from TE is not modelled.

A Key notion in the implementation is the *event loop*. An event loop is a program part that essentially comes down to the process depicted in Figure 12. An event loop can handle a set of events, this set corresponds to the notion of event state in the specification. However, in the implementation no message is communicated that corresponds to a `STATE-MESSAGE`. In the implementation the components TE and UI have more information about the state of UI and SE, TE and SE respectively. As a result, UI and TE can, and are allowed to, change their own event state themselves. So, there is no need for communication pairs also, i.e., the response to an event is not needed. E.g., when the Quit button is pushed, UI knows it should disable this button, whereas in the specification UI is explicitly instructed to do so. Components adjust the set of acceptable events in their event loop by recursive or tail recursive call of the event loop with a possibly new set of acceptable events.

7 Concluding remarks

In this report an approach is presented in which communication between editor components, and between editor components of the ASF+SDF Meta-environment and a user, can be modelled in a structured way. This approach is laid down in the interaction model. Furthermore the Quit, Parse and Expand interactions are modelled by means of interworkings and the editor as a whole, with emphasis on the interactions, is also modelled in PSF. We have the idea that the efforts on the interaction model and the specifications helped in finding a simple and reliable implementation of the interactions between the user and the three editor components. It is our belief that this is due to the fact that the interworkings and PSF enable one to model a solution in an abstract, but still very down to earth, manner. As result the project showed that formal specification techniques, here PSF and interworkings, can successfully be put to work.

We have no idea how expressive the interaction model is. Can the interaction model be used to model, in a natural way, interactions between more and other ASF+SDF Meta-environment components? Questions like this one call for a more structured and abstract approach to design of interactions models as such, whereas in this paper an interaction model is only part of the study. See in this respect work on the ToolBus [BKli94].

It is clear that the interaction model proposed in this paper has its shortcomings. By simulation of the specification a deadlock was found when the user, in some order, quickly clicked the Quit button of UI and typed 'ESC P' to TE. It turned out that the implementation contained this error too. The deadlock can easily be circumvented in a specification without disobeying the interaction model: route all communication with the user via UI. However, from an implementation point of view this is not an acceptable solution (much overhead). This leads at least at this point to a topological difference between routing of messages in the specification and in the implementation. Maybe the interaction model is not expressive enough. The deadlock found is not special, it can be characterised as as a deadlock due to race conditions, see e.g. [T84]. We expect race conditions to occur often in a system of multiple cooperating components and with a user that interfaces to more than one of these components, and suggest to pay extra attention to the avoidance of race conditions in the design of these systems. In this report the 'ESC P' is simply left out.

About the PSF specification itself, we think the communication atoms are hard to read. The part on `STATE-MESSAGE` can be hard to grasp. Furthermore, there should be axioms added to express the relation between the operations on `STATE-MESSAGE`.

The interworkings have been a handy specification means to sketch communication behaviour in. In discussions on the communication behaviour of the editor interworkings formed an important medium. We suggest to use interworkings in the technical documentation for the editor.

Acknowledgements

The idea to specify the interaction between the editor components in PSF originates with Paul Klint, Wilco Koorn and Jan Bergstra. Wilco Koorn and Huub Bakker helped us very much in understanding the intricacies of the editor. Mark van Wijk made the Interworking Toolset available to us. He also did some programming on these tools that made it possible to generate interworkings in the EPS-format.

References

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, 1990.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing Science*, 3(2):142–188, 1991.

[BB93] J.C.M. Baeten and J.A. Bergstra. Real space process algebra. *Journal of Formal Aspects of Computing Science*, 5(6):481–529, 1993.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[BK93] H.C.N. Bakker and J.W.C. Koorn. *Building an Editor from Existing Components: an Exercise in Software Re-use.* Report P9312, Programming Research Group, University of Amsterdam, 1993.

[BHK89a] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In *Algebraic specification* [BHK89a], pages 1–66.

[BHK89c] J.A. Bergstra, J. Heering, and P. Klint. A simple programming language and its implementation. In *Algebraic specification* [BHK89a], pages 67–132.

[BKli94] J.A. Bergstra and P. Klint. *The ToolBus – a Component Interconnection Architecture–.* P9408, Programming Research Group, University of Amsterdam, Amsterdam, 1994.

[Bru93] J.J. Brunekreef. Two simple protocols for local area networks. In Mauw and Veltink [MV93], pages 135–160.

[Epo92] University of Illinois, Urbana-Champaign. *Epoch, GNU Emacs for the X Windowing System*, 1992. Release 4.0, based on GNU Emacs 18.58.

[FvW92] W.J. Fokkink and J.J. van Wamel. *An algebraic specification of interest products.* Confidential Report CR9201, Programming Research Group, University of Amsterdam, Amsterdam, 1992.

[H93] J.A. Hillebrand. *An algebraic specification of a manufacturing system with hierarchical control.* Report P9331, Programming Research Group, University of Amsterdam, 1993.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[K93] P. Klint. A meta-environment for generating programming environments. In *ACM Transactions on Software Engineering and Methodology*, 2(2): 176-201, 1993.

[Koo92] J.W.C. Koorn. A generic text and structure editor. In J.L.G. Dietz, ed., *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pp. 168-177. SION, 1992.

[Koo94] J. W. C. Koorn. *Generating uniform user-interfaces for interactive programming environments.* PhD. Thesis, University of Amsterdam, 1994.

[M91] S. Mauw. *PSF, a process specification formalism.* PhD. Thesis, University of Amsterdam 1991.

[MV89] S. Mauw and G.J. Veltink. An introduction to PSFd. In J. Diaz, F. Orejas eds. *Proc. International Joint Conference on Theory and Practice of Software Development, TAPSOFT '89*, LNCS 352, pp. 272-285, Springer Verlag, 1989.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae* XIII, pp. 85-139, 1990.

[MV93] S. Mauw and G.J. Veltink. *Algebraic Specification of Communication Protocols.* Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[MWW92] S. Mauw, M. van Wijk and T. Winter. *Syntax and Semantics of Synchronous Interworkings.* Philips Research RWB508RE922436, 1992.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[Pon92] A. Ponse. *Process algebra with data.* PhD. thesis, University of Amsterdam, 1992.

[PV93] A. Ponse and J.A. Verschuren. *An algebraic specification of a model factory, part III.* Report P9303, Programming Research Group, University of Amsterdam, 1993.

[T84] A.S. Tanenbaum. *Structured Computer Organization.* Prentice–Hall, 1984.

[Vel93] G.J. Veltink. *The PSF toolkit.* Computer Networks and ISDN Systems, 25:875–898, 1993.

[VW93] S.F.M. van Vlijmen and A. van Waveren. *On Generating Synchronous Interworkings from PSF Process Traces.* Report P9304, Programming Research Group, University of Amsterdam, 1993.

[W92] J.J. van Wamel. *A study of a one bit sliding window protocol.* Report P9212, Programming Research Group, University of Amsterdam, 1992.

[W93] J.J. van Wamel. *A Library for PSF.* Report P9301, Programming Research Group, University of Amsterdam, 1993.

A Interworkings for the Quit interaction

In this appendix the interworkings for the Quit interaction are depicted. In Section 2.1 was explained that the macro facility of the interworking language is misused to 'model' choice and recursion. On top of that, in Figure 13 this information on order, choices and recursion for the Quit interworkings is depicted also. Execution starts in Quit 1. Next there is a choice between Quit 2 and Quit 3 and so on. Still the interworkings only model a part of the editor. How the execution at Quit 1 is started and which process takes over after, e.g., Quit 5 is not modelled.

Names like 'Quit 1' and 'Quit 2' are used in comments of module `Structure-Editor` to relate the PSF specification and the interworkings in this appendix. There are some mismatches between interworkings and specification text mainly due to the fact that in the interworkings the processes TE, UI, USER and SE are depicted together, whereas processes TE, UI and SE reside in different PSF modules and USER is not modelled at all. Typical mismatches are e.g.: comment `-- Quit 1` in module `Structure-Editor` denotes the beginning of the handler for `QUIT`, whereas in interworking Quit 1 not the first but the third action corresponds to the beginning of the handler.

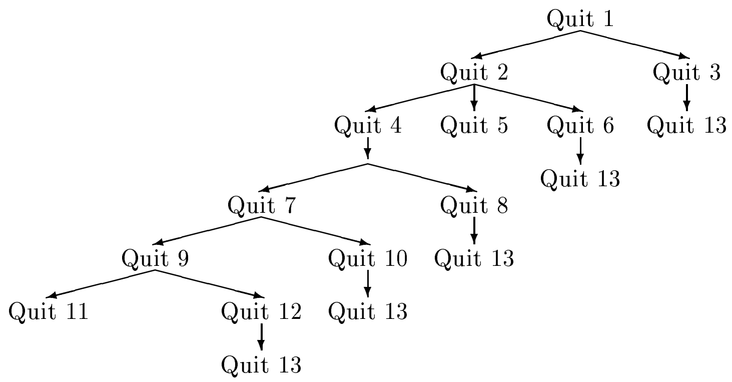


Figure 13: Relation of the various interworkings for the Quit interaction

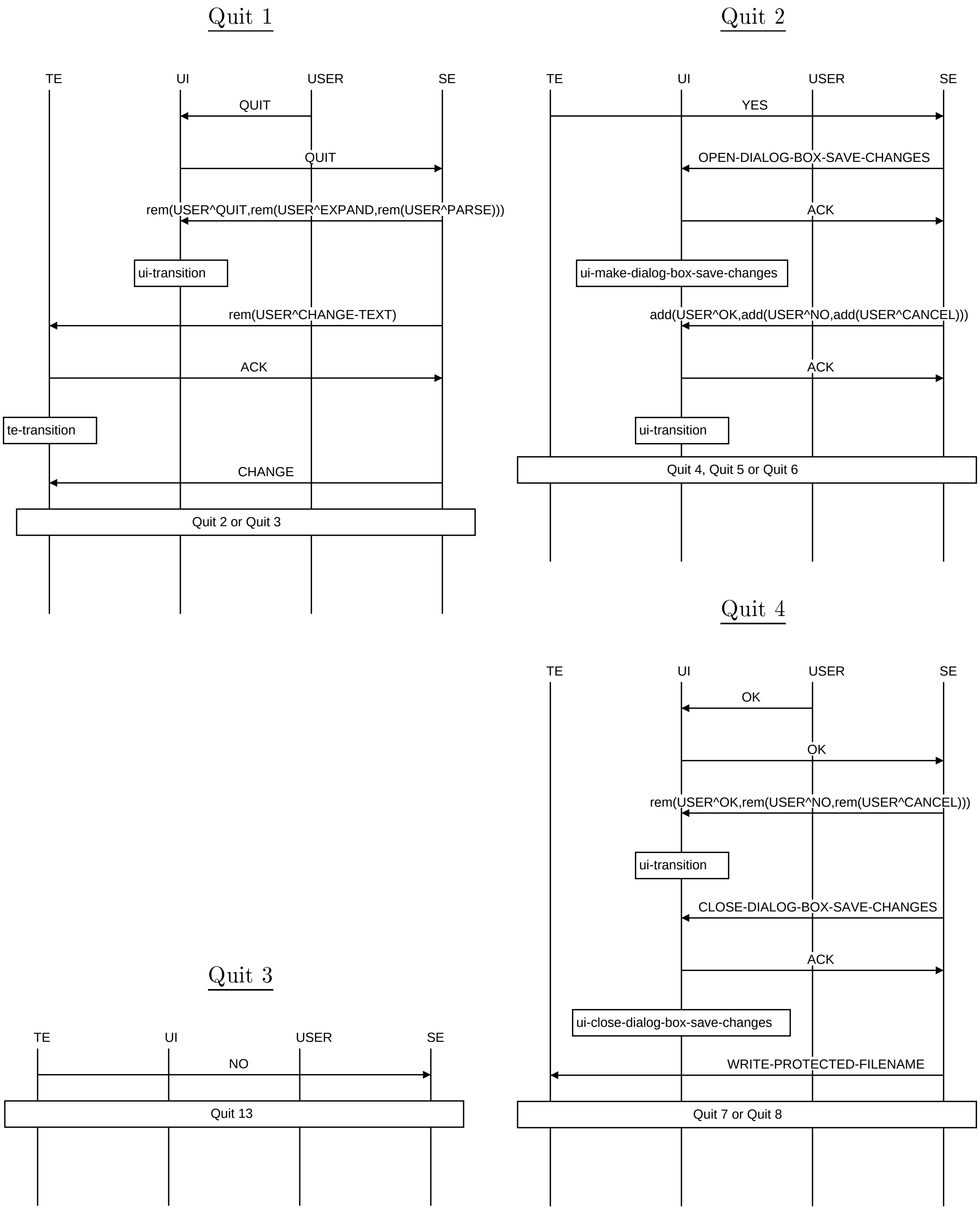
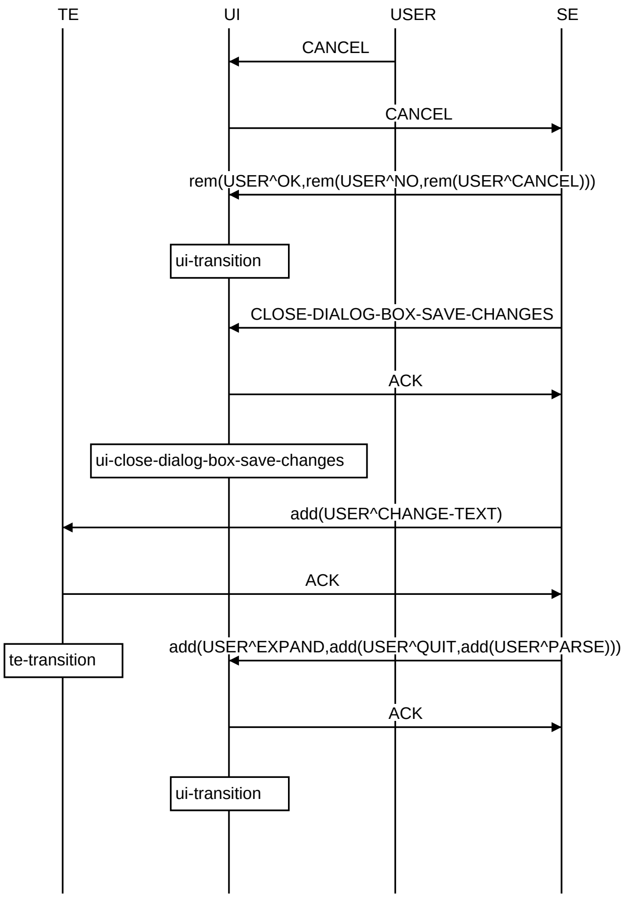
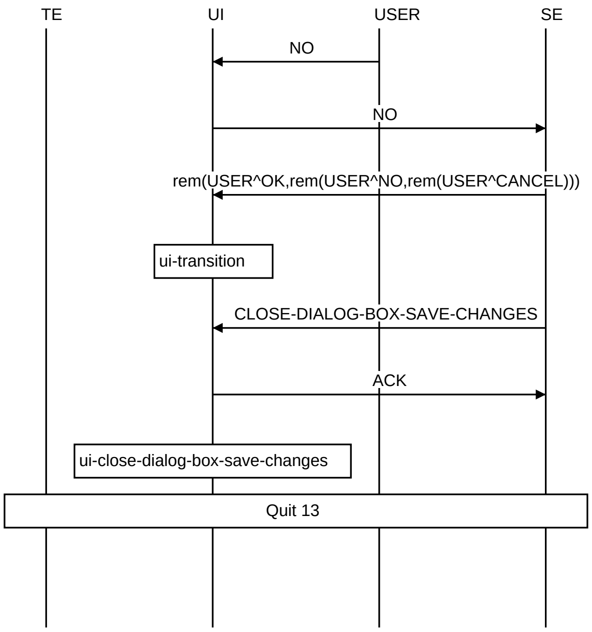


Figure 14: Quit 1, Quit 2, Quit 3, Quit 4

Quit 5



Quit 6



Quit 7

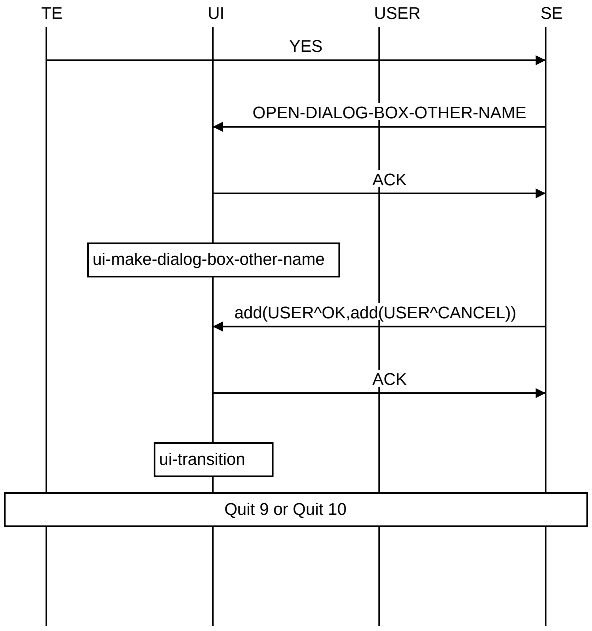
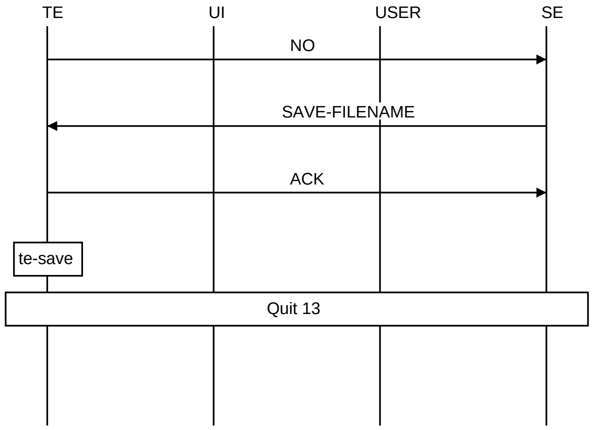
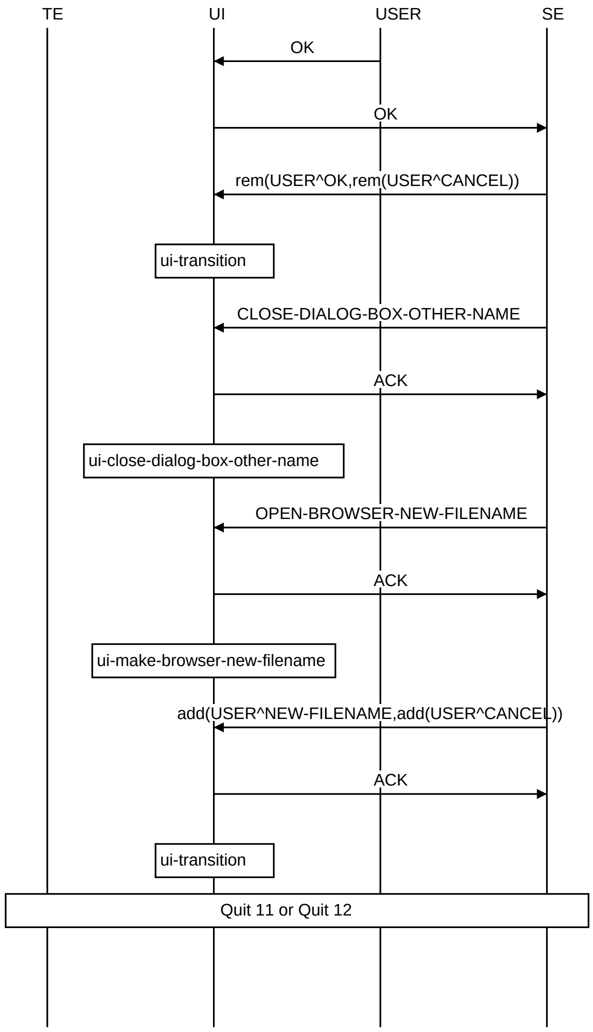


Figure 15: Quit 5, Quit 6, Quit 7

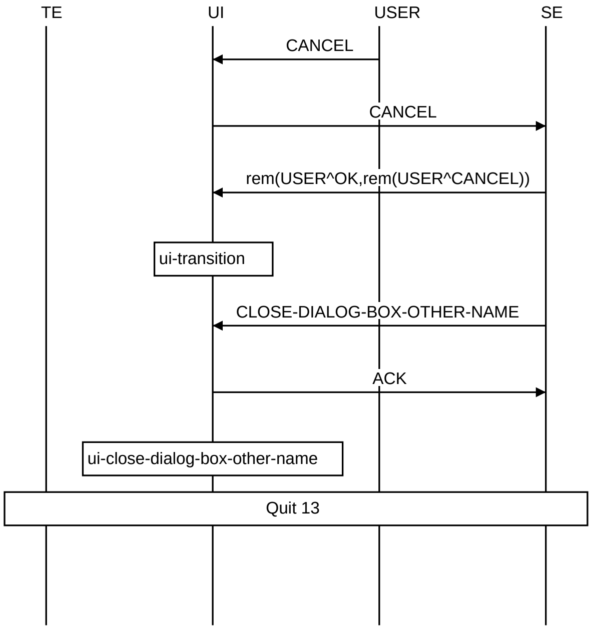
Quit 8



Quit 9



Quit 10



Quit 11

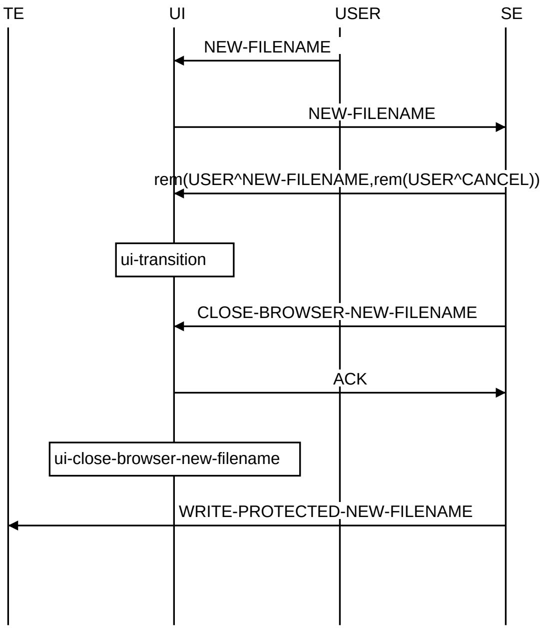
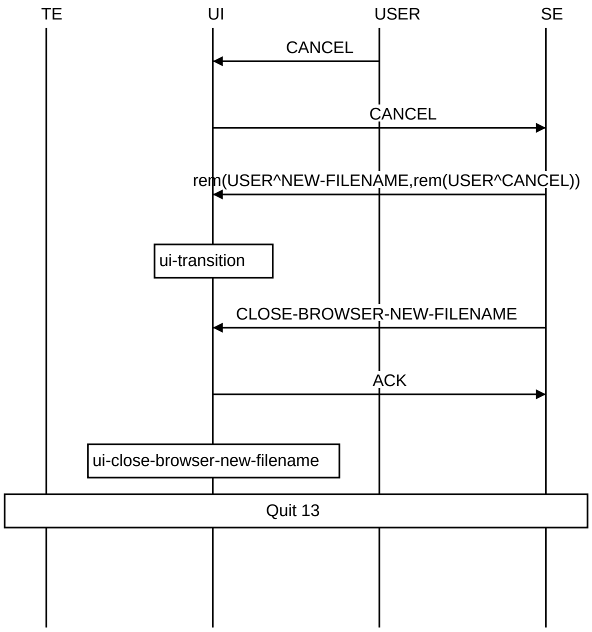


Figure 16: Quit 8, Quit 9, Quit 10, Quit 11

Quit 12



Quit 13

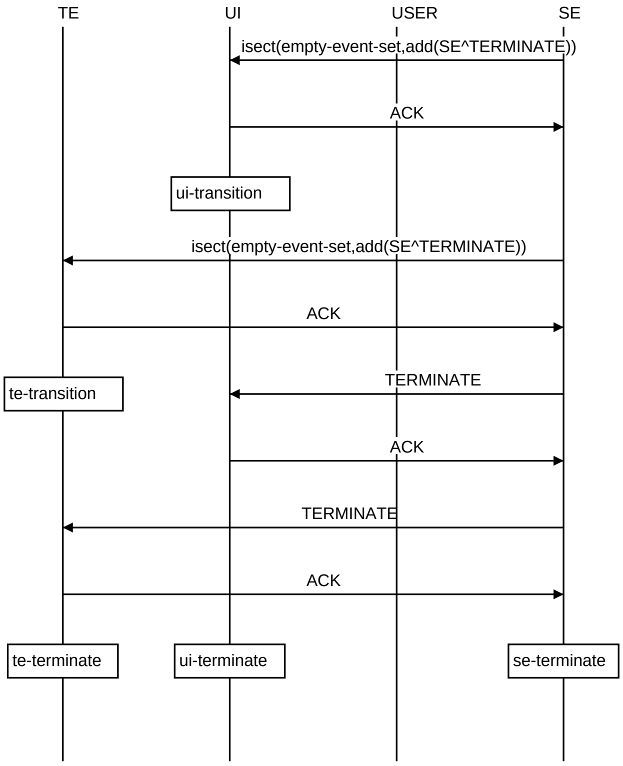


Figure 17: Quit 12, Quit 13

B Interworkings for the Parse interaction

In this appendix the interworkings modelling the Parse interaction are given. The names of the interworkings 'Parse 1', 'Parse 2' etc. are used to link the interworkings to the text of module `Structure-Editor`.

After interworking Parse 5 there should strictly follow three interworkings. One for handling `CHOICE1`, one for handling `CHOICE2` and one for handling `CANCEL`. The first two are combined for simplicity by introducing a message `CHOICEi` where `i` is 1 or 2.

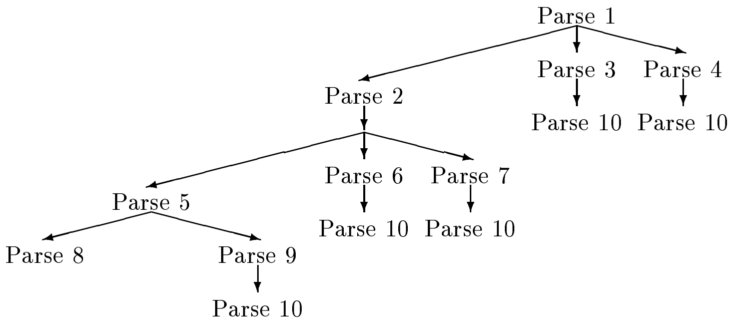


Figure 18: Relation of the various interworkings for the Parse interaction

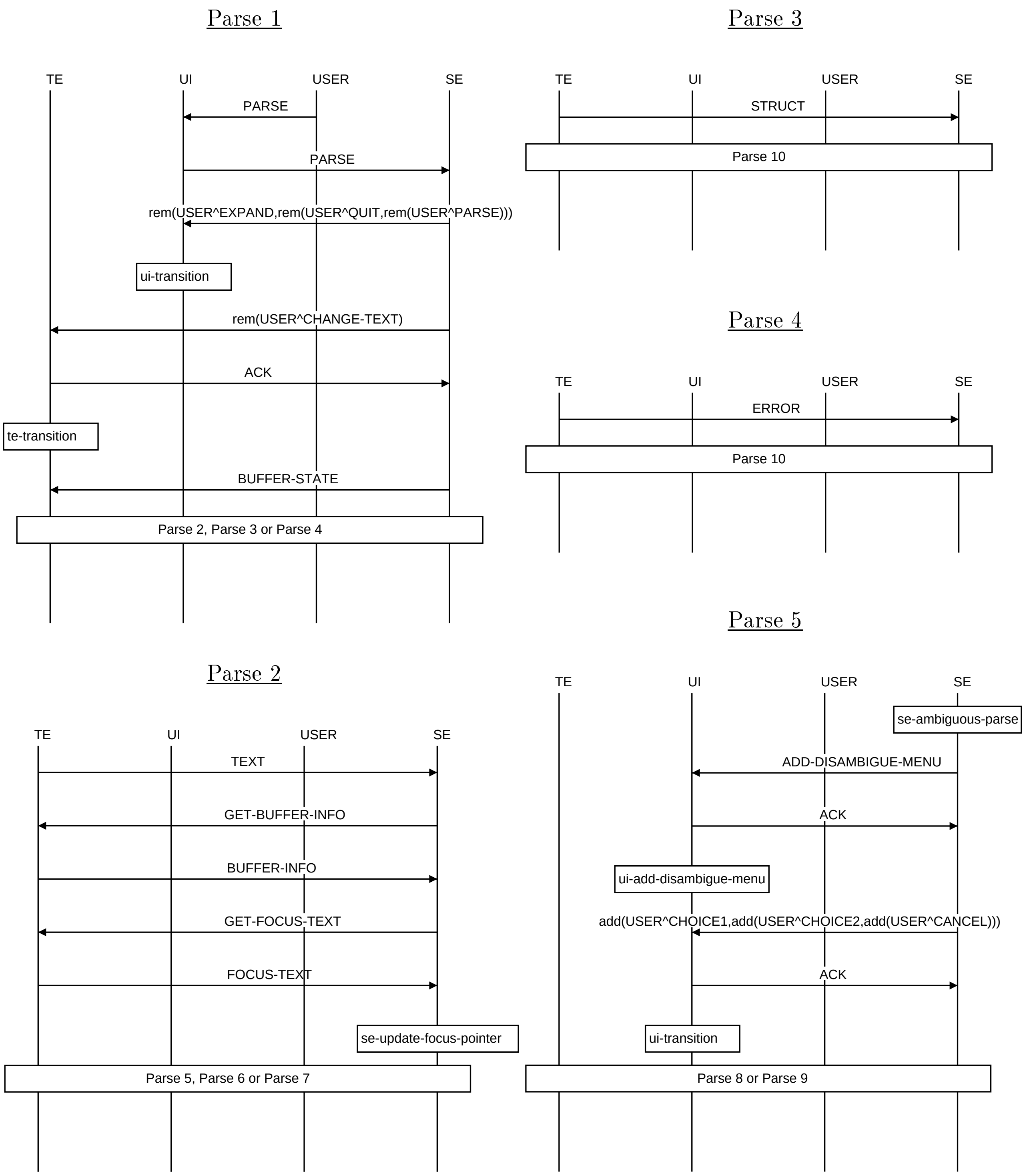


Figure 19: Parse 1, Parse 2, Parse 3, Parse 4, Parse 5

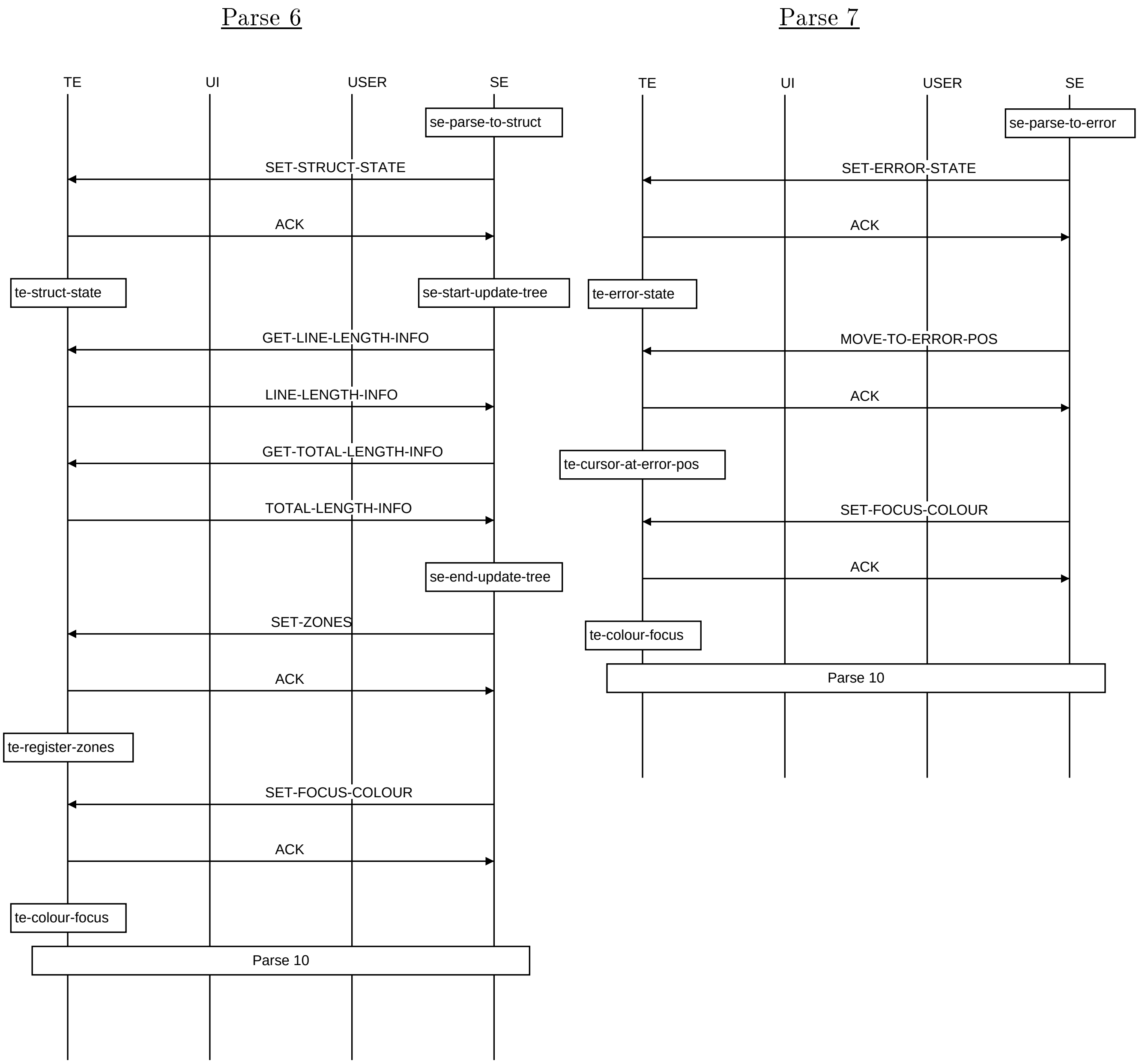
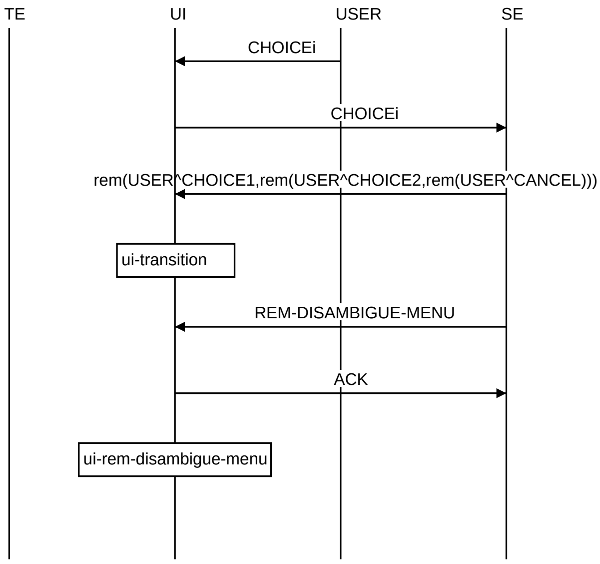
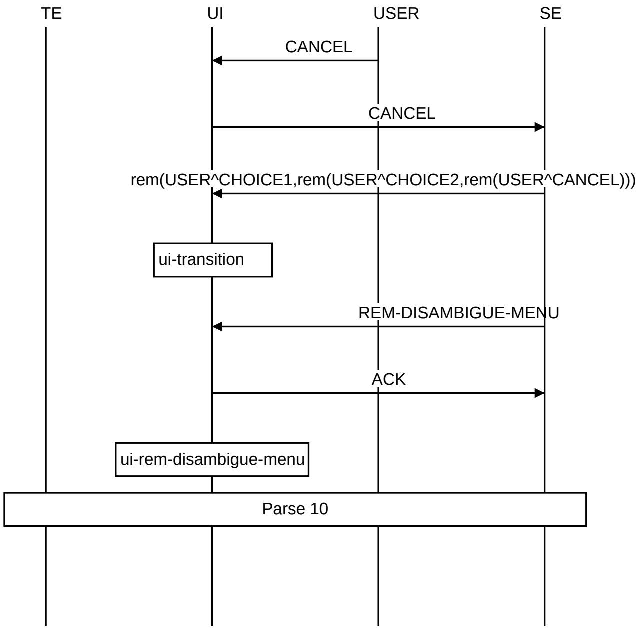


Figure 20: Parse 6, Parse 7

Parse 8



Parse 9



Parse 10

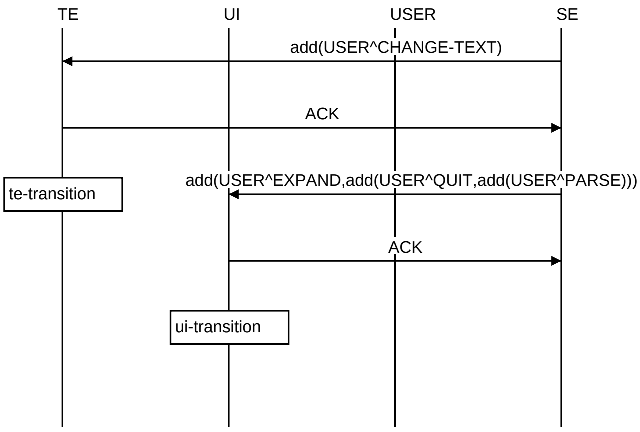


Figure 21: Parse 8, Parse 9, Parse 10

C Interworkings for the Expand interaction

In interworking Expand 6, `CHOICEi` is used to model the handling for `CHOICE1` and `CHOICE2` in one interworking. Names like 'Expand 1' and 'Expand 2' are used in comments in module `Structure-Editor` to link the interworkings to the specification in PSF.

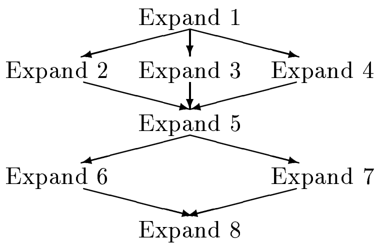


Figure 22: Relation of the various interworkings for the Expand interaction

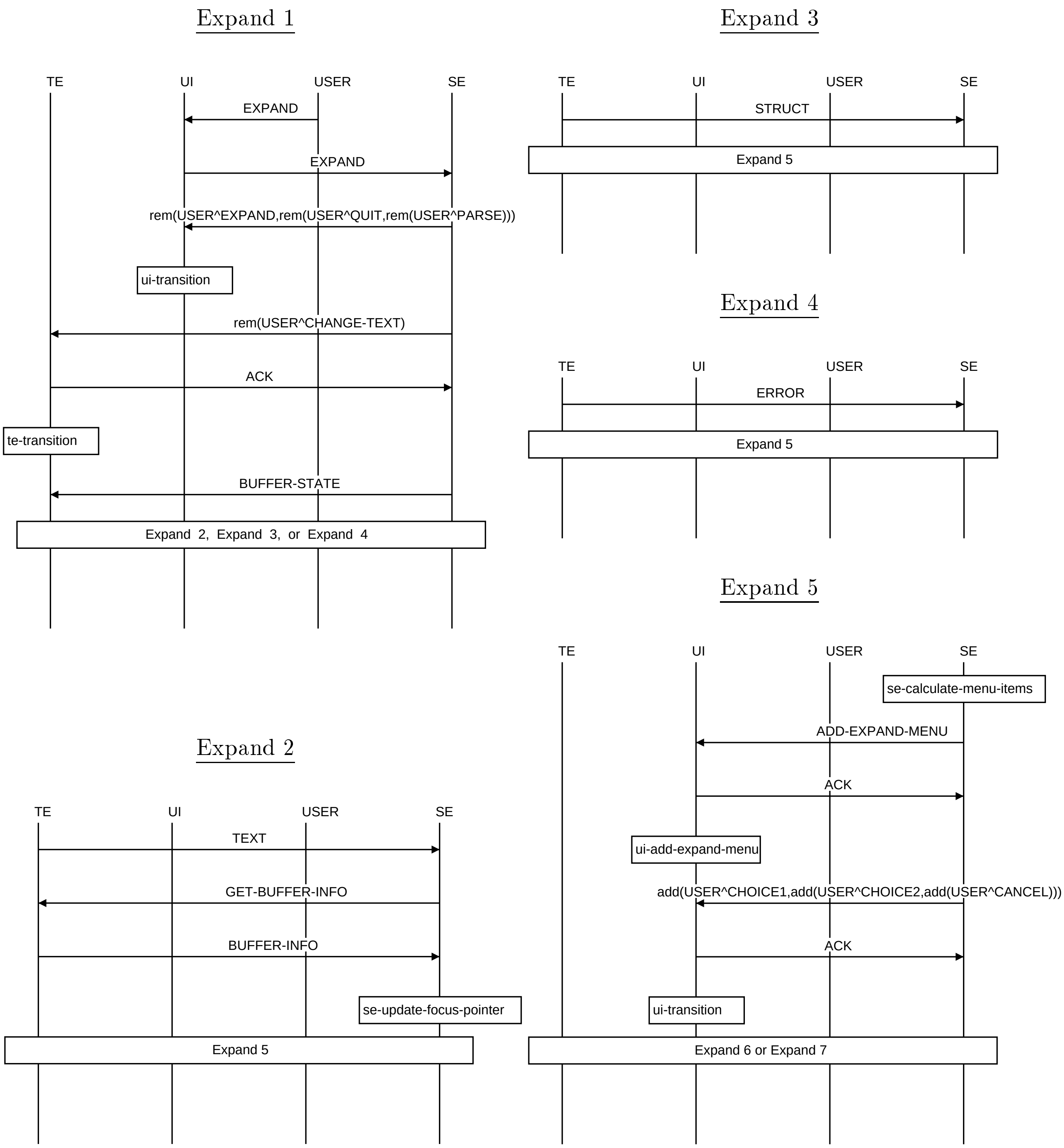


Figure 23: Expand 1, Expand 2, Expand 3, Expand 4, Expand 5

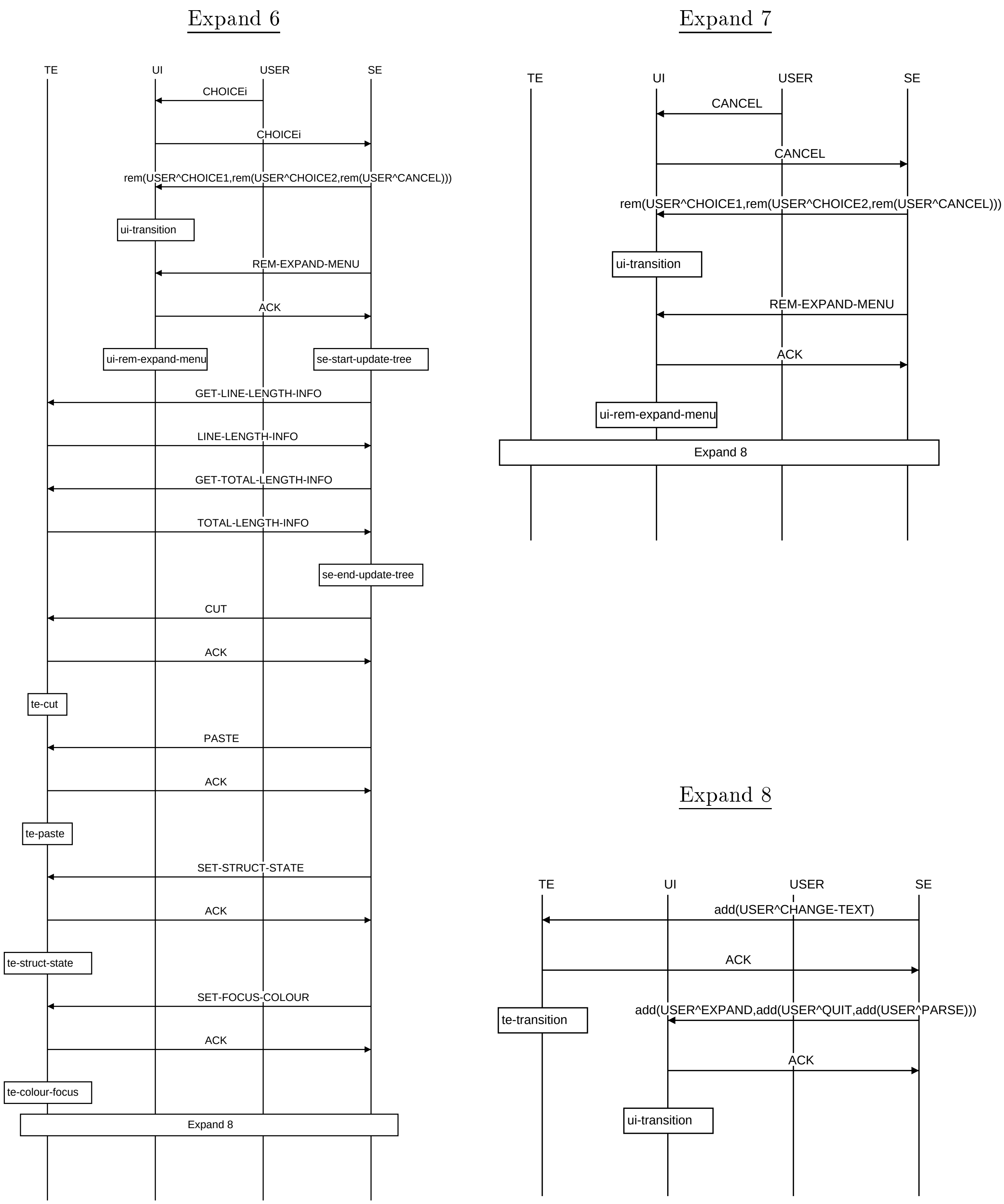


Figure 24: Expand 6, Expand 7, Expand 8

D The specification

In Figure 25 the import structure of the specification is depicted. Note that the modules imported from the PSF library [W93] are not listed. The comments in module `Structure-Editor` relate the specification text to the interworkings in appendices A, B and C.

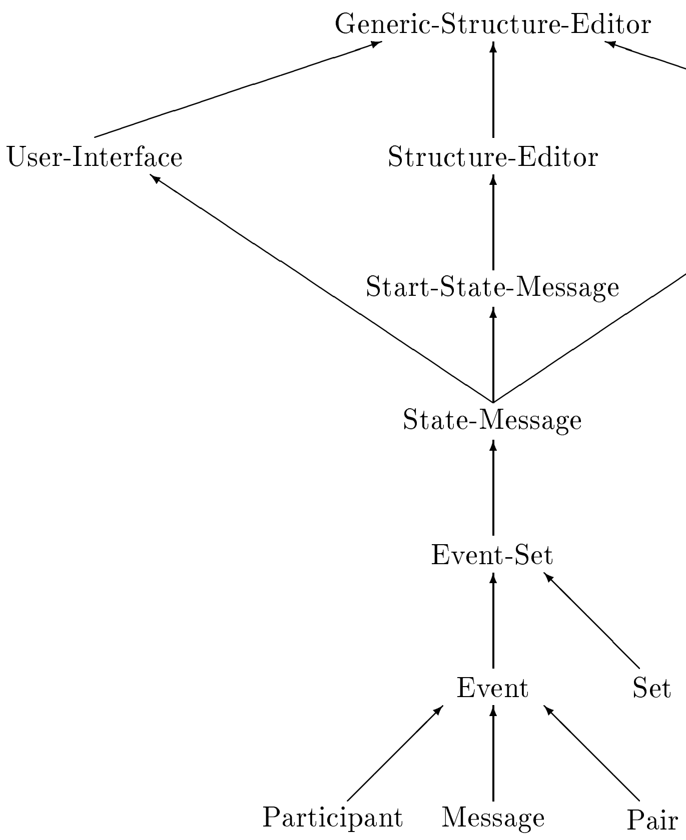


Figure 25: Import structure of the specification, except the import of Booleans and Naturals

```
data module Buffer-State
begin
  exports
    begin
      sorts
        BUFF-STATE
      functions
        text-state   : -> BUFF-STATE
        struct-state : -> BUFF-STATE
        error-state  : -> BUFF-STATE
        eq           : BUFF-STATE # BUFF-STATE -> BOOLEAN
    end
  imports
    Naturals
  functions
    i                : BUFF-STATE -> NATURAL
  variables
    m, m' : -> BUFF-STATE
  equations
  [i1] i(text-state)   = nat(^0)
  [i2] i(struct-state) = nat(^1)
  [i3] i(error-state)  = nat(^2)
  [e1] eq(m,m')        = eq(i(m),i(m'))
end Buffer-State

data module Participant
begin
  exports
    begin
      sorts
        PARTICIPANT
      functions
        TE   : -> PARTICIPANT
        SE   : -> PARTICIPANT
        UI   : -> PARTICIPANT
        USER : -> PARTICIPANT
        eq   : PARTICIPANT # PARTICIPANT -> BOOLEAN
    end
  imports
    Naturals
  functions
    i        : PARTICIPANT -> NATURAL
  variables
    pn, pn'  : -> PARTICIPANT
  equations
  [i1] i(TE)      = nat(^0)
  [i2] i(SE)      = nat(^1)
  [i3] i(UI)      = nat(^2)
  [i4] i(USER)    = nat(^3)
  [e1] eq(pn,pn') = eq(i(pn),i(pn'))
end Participant

data module Message
begin
  exports
    begin
      sorts
        MESSAGE
      functions
        OPEN-DIALOG-BOX-SAVE-CHANGES  : -> MESSAGE
        CLOSE-DIALOG-BOX-SAVE-CHANGES : -> MESSAGE
        QUIT                          : -> MESSAGE
        WINDOW-RESIZE                 : -> MESSAGE
        YES                           : -> MESSAGE
        OK                            : -> MESSAGE
        NO                            : -> MESSAGE
        CANCEL                        : -> MESSAGE
        SAVE-FILENAME                 : -> MESSAGE
        CHANGE                        : -> MESSAGE
        CHANGE-TEXT                   : -> MESSAGE
        ACK                           : -> MESSAGE
        WRITE-PROTECTED-FILENAME      : -> MESSAGE
        OPEN-DIALOG-BOX-OTHER-NAME    : -> MESSAGE
        CLOSE-DIALOG-BOX-OTHER-NAME   : -> MESSAGE
        OPEN-BROWSER-NEW-FILENAME     : -> MESSAGE
        CLOSE-BROWSER-NEW-FILENAME    : -> MESSAGE
        NEW-FILENAME                  : -> MESSAGE
        WRITE-PROTECTED-NEW-FILENAME  : -> MESSAGE
        TERMINATE                     : -> MESSAGE
        EXPAND                        : -> MESSAGE
        BUFFER-STATE                  : -> MESSAGE
        TEXT                          : -> MESSAGE
        ERROR                         : -> MESSAGE
        STRUCT                        : -> MESSAGE
        GET-FOCUS-TEXT                : -> MESSAGE
        FOCUS-TEXT                    : -> MESSAGE
        ADD-EXPAND-MENU               : -> MESSAGE
        REM-EXPAND-MENU               : -> MESSAGE
        CHOICE1                       : -> MESSAGE
        CHOICE2                       : -> MESSAGE
        GET-LINE-LENGTH-INFO          : -> MESSAGE
        LINE-LENGTH-INFO              : -> MESSAGE
        GET-TOTAL-LENGTH-INFO         : -> MESSAGE
        TOTAL-LENGTH-INFO             : -> MESSAGE
        CUT                           : -> MESSAGE
        PASTE                         : -> MESSAGE
        SET-STRUCT-STATE              : -> MESSAGE
        SET-ERROR-STATE               : -> MESSAGE
        PARSE                         : -> MESSAGE
        GET-BUFFER-INFO               : -> MESSAGE
        BUFFER-INFO                   : -> MESSAGE
        SET-FOCUS-COLOUR              : -> MESSAGE
        SET-ZONES                     : -> MESSAGE
        MOVE-TO-ERROR-POS             : -> MESSAGE
        ADD-DISAMBIGUE-MENU           : -> MESSAGE
        REM-DISAMBIGUE-MENU           : -> MESSAGE
        eq                            : MESSAGE # MESSAGE -> BOOLEAN
    end
  imports
    Naturals
  functions
    i       : MESSAGE -> NATURAL
  variables
    sm, sm' : -> MESSAGE
  equations
  [i1]  i(OPEN-DIALOG-BOX-SAVE-CHANGES)  = nat(^0)
  [i2]  i(QUIT)                          = nat(^1)
  [i3]  i(WINDOW-RESIZE)                 = nat(^2)
  [i4]  i(YES)                           = nat(^3)
  [i5]  i(NO)                            = nat(^4)
  [i6]  i(SAVE-FILENAME)                 = nat(^5)
  [i7]  i(CHANGE)                        = nat(^6)
  [i8]  i(ACK)                           = nat(^7)
  [i9]  i(TERMINATE)                     = nat(^8)
  [i10] i(CHANGE-TEXT)                   = nat(^9)
  [i11] i(CANCEL)                        = nat(^1^0)
  [112] i(WRITE-PROTECTED-FILENAME)      = nat(^1^1)
  [i13] i(OPEN-DIALOG-BOX-OTHER-NAME)    = nat(^1^2)
  [i14] i(OPEN-BROWSER-NEW-FILENAME)     = nat(^1^3)
  [i15] i(NEW-FILENAME)                  = nat(^1^4)
  [i16] i(WRITE-PROTECTED-NEW-FILENAME)  = nat(^1^5)
  [117] i(CLOSE-DIALOG-BOX-SAVE-CHANGES) = nat(^1^6)
  [i18] i(CLOSE-DIALOG-BOX-OTHER-NAME)   = nat(^1^7)
  [i19] i(CLOSE-BROWSER-NEW-FILENAME)    = nat(^1^8)
  [i20] i(OK)                            = nat(^1^9)
  [i21] i(EXPAND)                        = nat(^2^0)
  [i22] i(BUFFER-STATE)                  = nat(^2^1)
  [i23] i(TEXT)                          = nat(^2^2)
  [i24] i(ERROR)                         = nat(^2^3)
  [i25] i(STRUCT)                        = nat(^2^4)
  [i26] i(GET-FOCUS-TEXT)                = nat(^2^5)
  [i27] i(FOCUS-TEXT)                    = nat(^2^6)
  [i28] i(ADD-EXPAND-MENU)               = nat(^2^7)
  [i29] i(REM-EXPAND-MENU)               = nat(^2^8)
  [i30] i(CHOICE1)                       = nat(^2^9)
  [i31] i(CHOICE2)                       = nat(^3^0)
```

```
  [i32] i(GET-LINE-LENGTH-INFO)          = nat(^3^1)
  [i33] i(LINE-LENGTH-INFO)              = nat(^3^2)
  [i34] i(GET-TOTAL-LENGTH-INFO)         = nat(^3^3)
  [i35] i(TOTAL-LENGTH-INFO)             = nat(^3^4)
  [i36] i(CUT)                           = nat(^3^5)
  [i37] i(PASTE)                         = nat(^3^6)
  [i38] i(SET-STRUCT-STATE)              = nat(^3^7)
  [i49] i(PARSE)                         = nat(^3^8)
  [i40] i(GET-BUFFER-INFO)               = nat(^3^9)
  [i41] i(BUFFER-INFO)                   = nat(^4^0)
  [i42] i(SET-FOCUS-COLOUR)              = nat(^4^1)
  [i43] i(MOVE-TO-ERROR-POS)             = nat(^4^2)
  [i44] i(SET-ERROR-STATE)               = nat(^4^3)
  [i45] i(ADD-DISAMBIGUE-MENU)           = nat(^4^4)
  [i46] i(REM-DISAMBIGUE-MENU)           = nat(^4^5)
  [i47] i(SET-ZONES)                     = nat(^4^6)
  [e1]  eq(sm,sm') = eq(i(sm),i(sm'))
end Message

data module Pair
begin
  parameters Item1
    begin
      sorts
        ITEM1
      functions
        eq   : ITEM1 # ITEM1 -> BOOLEAN
    end Item1,
    Item2
    begin
      sorts
        ITEM2
      functions
        eq   : ITEM2 # ITEM2 -> BOOLEAN
    end Item2
  exports
    begin
      sorts
        PAIR
      functions
        _^_    : ITEM1 # ITEM2 -> PAIR
        proj1  : PAIR          -> ITEM1
        proj2  : PAIR          -> ITEM2
        eq     : PAIR # PAIR   -> BOOLEAN
    end
  imports Booleans
  variables
    i1,i1' :-> ITEM1
    i2,i2' :-> ITEM2
  equations
  [P1] proj1(i1^i2) = i1
  [P2] proj2(i1^i2) = i2
  [e1] eq(i1^i2,i1'^i2') = and(eq(i1,i1'), eq(i2,i2'))
end Pair

data module Event
begin
  imports
    Pair {Item1 bound by [ITEM1 -> PARTICIPANT, eq -> eq] to Participant
          Item2 bound by [ITEM2 -> MESSAGE    , eq -> eq] to Message
              renamed by [PAIR  -> EVENT                ]}
end Event

data module Set
begin
  parameters
    Item begin
      sorts
        ITEM
      functions
        eq      : ITEM # ITEM -> BOOLEAN
    end Item
  exports
    begin
      sorts
        SET
      functions
        empty   :              -> SET
        add     : ITEM # SET -> SET
        is-in   : ITEM # SET -> BOOLEAN
        rem     : ITEM # SET -> SET
        union   : SET # SET  -> SET
        isect   : SET # SET  -> SET
        diff    : SET # SET  -> SET
        subset  : SET # SET  -> BOOLEAN
        eq      : SET # SET  -> BOOLEAN
    end
  imports
    Naturals
  variables
    i, i' : -> ITEM
    s, s' : -> SET
  equations
    [a1]  add(i, add(i', s))    = add(i', add(i, s))
    [a2]  add(i, add(i, s))     = add(i, s)

    [ii1] is-in(i, empty)       = false
    [ii2] is-in(i, add(i', s))  = or(eq(i, i'), is-in(i, s))

    [R1]  rem(i, empty)         = empty
    [R2]  rem(i, add(i, s))     = rem(i, s)
    [R3]  rem(i, add(i', s))    = add(i', rem(i, s)) when eq(i, i') = false

    [U1]  union(empty, s)       = s
    [U2]  union(add(i, s), s')  = add(i, union(s, s'))

    [i1]  isect(empty, s)       = empty
    [i2]  isect(add(i, s), s')  = add(i, isect(s, s')) when is-in(i, s') = true
    [i3]  isect(add(i, s), s')  = isect(s, s') when is-in(i, s') = false

    [D1]  diff(s, empty)        = s
    [D2]  diff(s, add(i, s'))   = rem(i, diff(s, s'))

    [S1]  subset(empty, s)      = true
    [S2]  subset(add(i, s), s') = and(is-in(i, s'), subset(s, s'))

    [eq1] eq(s, s')             = and(subset(s, s'), subset(s', s))
end Set

data module Event-Set
begin
  imports
    Set {Item bound by [ITEM -> EVENT    , eq    -> eq ] to Event
            renamed by [SET  -> EVENT-SET, empty -> empty-event-set]}
end Event-Set

data module State-Message
begin
  exports
    begin
      sorts
        STATE-MESSAGE
      functions
        union : EVENT-SET                 -> STATE-MESSAGE
        diff  : EVENT-SET                 -> STATE-MESSAGE
        isect : EVENT-SET                 -> STATE-MESSAGE
        add   : EVENT                     -> STATE-MESSAGE
        rem   : EVENT                     -> STATE-MESSAGE
        union : EVENT-SET # STATE-MESSAGE -> STATE-MESSAGE
        diff  : EVENT-SET # STATE-MESSAGE -> STATE-MESSAGE
        isect : EVENT-SET # STATE-MESSAGE -> STATE-MESSAGE
        add   : EVENT     # STATE-MESSAGE -> STATE-MESSAGE
        rem   : EVENT     # STATE-MESSAGE -> STATE-MESSAGE
        apply : STATE-MESSAGE # EVENT-SET -> EVENT-SET
    end
  imports
```

```
    Event-Set
  variables
    sm        : -> EVENT
    sms, sms' : -> EVENT-SET
    em        : -> STATE-MESSAGE
  equations
  [ap1]  apply(union(sms),sms')    = union(sms,sms')
  [ap2]  apply(union(sms,em),sms') = apply(em,union(sms,sms'))
  [ap3]  apply(diff(sms),sms')     = diff(sms',sms)
  [ap4]  apply(diff(sms,em),sms')  = apply(em,diff(sms',sms))
  [ap5]  apply(isect(sms),sms')    = isect(sms,sms')
  [ap6]  apply(isect(sms,em),sms') = apply(em,isect(sms,sms'))
  [ap7]  apply(add(sm),sms')       = add(sm,sms')
  [ap8]  apply(add(sm,em),sms')    = apply(em,add(sm,sms'))
  [ap9]  apply(rem(sm),sms')       = rem(sm,sms')
  [ap10] apply(rem(sm,em),sms')    = apply(em,rem(sm,sms'))
end State-Message

process module User-Interface
begin
  exports
    begin
      atoms
        event-in-UI     : PARTICIPANT # MESSAGE
        response-in-UI  : PARTICIPANT # MESSAGE
        event-out-UI    : PARTICIPANT # MESSAGE
        response-out-UI : PARTICIPANT # MESSAGE
        event-in-UI     : PARTICIPANT # STATE-MESSAGE
        response-in-UI  : PARTICIPANT # STATE-MESSAGE
        ui-terminate
        ui-transition
        ui-window-resize
        ui-make-dialog-box-save-changes
        ui-close-dialog-box-save-changes
        ui-make-dialog-box-other-name
        ui-close-dialog-box-other-name
        ui-make-browser-new-filename
        ui-close-browser-new-filename
        ui-add-expand-menu
        ui-rem-expand-menu
        ui-add-disambigue-menu
        ui-rem-disambigue-menu
      processes
        UI
    end
  imports
    State-Message
  processes
    UI            : EVENT-SET
    HANDLE        : PARTICIPANT # MESSAGE # EVENT-SET
    HANDLE        : PARTICIPANT # STATE-MESSAGE # EVENT-SET
    Handle-ack-or-event-state-change : EVENT-SET
  variables
    E  : -> EVENT-SET
    m  : -> MESSAGE
    sm : -> STATE-MESSAGE
    p  : -> PARTICIPANT
  definitions
    UI = UI(empty-event-set)

    UI(E)
    = sum( p in PARTICIPANT,
           sum(m in MESSAGE, [is-in(p^m, E) = true]
               -> event-in-UI(p,m) . HANDLE(p,m,E))
           +
           sum(m in STATE-MESSAGE, event-in-UI(p,m) . HANDLE(p,m,E))
         )

    HANDLE(USER,QUIT,E)
    = event-out-UI(SE,QUIT) . Handle-ack-or-event-state-change(E)

    HANDLE(USER,EXPAND,E)
    = event-out-UI(SE,EXPAND) . Handle-ack-or-event-state-change(E)

    HANDLE(USER,PARSE,E)
    = event-out-UI(SE,PARSE) . Handle-ack-or-event-state-change(E)

    HANDLE(USER, NEW-FILENAME,E)
    = event-out-UI(SE,NEW-FILENAME) . Handle-ack-or-event-state-change(E)

    HANDLE(USER,WINDOW-RESIZE,E)
    = ui-window-resize .
      event-out-UI(TE,WINDOW-RESIZE) .
      response-in-UI(TE,ACK) .
      UI(E)

    HANDLE(SE,OPEN-DIALOG-BOX-SAVE-CHANGES,E)
    = response-out-UI(SE,ACK) . ui-make-dialog-box-save-changes . UI(E)

    HANDLE(SE,CLOSE-DIALOG-BOX-SAVE-CHANGES,E)
    = response-out-UI(SE,ACK) . ui-close-dialog-box-save-changes . UI(E)

    HANDLE(SE,OPEN-DIALOG-BOX-OTHER-NAME,E)
    = response-out-UI(SE,ACK) . ui-make-dialog-box-other-name . UI(E)

    HANDLE(SE,CLOSE-DIALOG-BOX-OTHER-NAME,E)
    = response-out-UI(SE,ACK) . ui-close-dialog-box-other-name . UI(E)

    HANDLE(SE,OPEN-BROWSER-NEW-FILENAME,E)
    = response-out-UI(SE,ACK) . ui-make-browser-new-filename . UI(E)

    HANDLE(SE,CLOSE-BROWSER-NEW-FILENAME,E)
    = response-out-UI(SE,ACK) . ui-close-browser-new-filename . UI(E)

    HANDLE(SE,ADD-EXPAND-MENU,E)
    = response-out-UI(SE,ACK) . ui-add-expand-menu . UI(E)

    HANDLE(SE,REM-EXPAND-MENU,E)
    = response-out-UI(SE,ACK) . ui-rem-expand-menu . UI(E)

    HANDLE(SE,ADD-DISAMBIGUE-MENU,E)
    = response-out-UI(SE,ACK) . ui-add-disambigue-menu . UI(E)

    HANDLE(SE,REM-DISAMBIGUE-MENU,E)
    = response-out-UI(SE,ACK) . ui-rem-disambigue-menu . UI(E)

    HANDLE(USER,OK,E)
    = event-out-UI(SE,OK) .  Handle-ack-or-event-state-change(E)

    HANDLE(USER,NO,E)
    = event-out-UI(SE,NO) . Handle-ack-or-event-state-change(E)

    HANDLE(USER,CANCEL,E)
    = event-out-UI(SE,CANCEL) . Handle-ack-or-event-state-change(E)

    HANDLE(USER,m,E)
    = [or(eq(m,CHOICE1),eq(m,CHOICE2)) = true] ->
      event-out-UI(SE,m) . Handle-ack-or-event-state-change(E)

    HANDLE(SE,TERMINATE,E)
    = response-out-UI(SE,ACK) . ui-terminate

    HANDLE(p,sm,E)
    = response-out-UI(p,ACK) .  ui-transition . UI(apply(sm,E))

    Handle-ack-or-event-state-change(E)
    = ( response-in-UI(SE,ACK) . UI(E)
        +
        sum(sm in STATE-MESSAGE,
            response-in-UI(SE,sm) . ui-transition .  UI(apply(sm,E)))
      )
end User-Interface

process module Text-Editor
begin
  exports
```

```
  begin
    atoms
      event-in-TE     : PARTICIPANT # MESSAGE
      response-in-TE  : PARTICIPANT # MESSAGE
      event-out-TE    : PARTICIPANT # MESSAGE
      response-out-TE : PARTICIPANT # MESSAGE
      event-in-TE     : PARTICIPANT # STATE-MESSAGE
      response-in-TE  : PARTICIPANT # STATE-MESSAGE
      te-window-resize
      te-save
      te-transition
      te-terminate
      te-write-protected
      te-not-write-protected
      te-change-text
      te-cut
      te-paste
      te-colour-focus
      te-register-zones
      te-cursor-at-error-pos
      te-struct-state
      te-error-state
      te-text-state
    processes
      TE
  end
imports
  State-Message, Buffer-State,
  Booleans {renamed by [BOOLEAN -> TEXT-CHANGE,
                        true -> text-changed, false -> text-not-changed]}
processes
  TE           : EVENT-SET   # TEXT-CHANGE # BUFF-STATE
  HANDLE       : PARTICIPANT # MESSAGE # EVENT-SET #
                 TEXT-CHANGE # BUFF-STATE
  HANDLE       : PARTICIPANT # STATE-MESSAGE # EVENT-SET #
                 TEXT-CHANGE # BUFF-STATE
variables
  E  : -> EVENT-SET
  sm : -> STATE-MESSAGE
  tc : -> TEXT-CHANGE
  es : -> BUFF-STATE
definitions
  TE = TE(empty-event-set,text-not-changed,text-state)

  TE(E,tc,es)
  = sum( p in PARTICIPANT,
         sum(m in MESSAGE, [is-in(p^m,E) = true]
             -> event-in-TE(p,m) . HANDLE(p,m,E,tc,es))
         +
         sum(m in STATE-MESSAGE, event-in-TE(p,m) . HANDLE(p,m,E,tc,es))
       )

  HANDLE(UI,WINDOW-RESIZE,E,tc,es)
  = response-out-TE(UI,ACK) . te-window-resize . TE(E,tc,es)

  HANDLE(SE,SAVE-FILENAME,E,tc,es)
  = response-out-TE(SE,ACK) . te-save . TE(E,text-not-changed,es)

  HANDLE(SE,CHANGE,E,text-not-changed,es)
  = response-out-TE(SE,NO) . TE(E,text-not-changed,es)

  HANDLE(SE,CHANGE,E,text-changed,es)
  = response-out-TE(SE,YES) .  TE(E,text-changed,es)

  HANDLE(SE,TERMINATE,E,tc,es)
  = response-out-TE(SE,ACK) . te-terminate

  HANDLE(USER,CHANGE-TEXT,E,tc,es)
  = te-change-text . te-text-state . TE(E,text-changed,text-state)

  HANDLE(SE,WRITE-PROTECTED-FILENAME,E,tc,es)
  = ( te-write-protected . response-out-TE(SE,YES)
      +
      te-not-write-protected . response-out-TE(SE,NO)
    ) .
    TE(E,tc,es)

  HANDLE(SE,BUFFER-STATE,E,tc,es)
  = ( [es = text-state]   -> response-out-TE(SE,TEXT)
      +
      [es = error-state]  -> response-out-TE(SE,ERROR)
      +
      [es = struct-state] -> response-out-TE(SE,STRUCT)
    ) .
    TE(E,tc,es)

  HANDLE(SE,GET-FOCUS-TEXT,E,tc,es)
  = response-out-TE(SE,FOCUS-TEXT) . TE(E,tc,es)

  HANDLE(SE,GET-BUFFER-INFO,E,tc,es)
  = response-out-TE(SE,BUFFER-INFO) . TE(E,tc,es)

  HANDLE(SE,GET-LINE-LENGTH-INFO,E,tc,es)
  = response-out-TE(SE,LINE-LENGTH-INFO) . TE(E,tc,es)

  HANDLE(SE,GET-TOTAL-LENGTH-INFO,E,tc,es)
  = response-out-TE(SE,TOTAL-LENGTH-INFO) . TE(E,tc,es)

  HANDLE(SE,CUT,E,tc,es)
  = response-out-TE(SE,ACK) . te-cut . TE(E,text-changed,es)

  HANDLE(SE,PASTE,E,tc,es)
  = response-out-TE(SE,ACK) . te-paste . TE(E,text-changed,es)

  HANDLE(SE,SET-STRUCT-STATE,E,tc,es)
  = response-out-TE(SE,ACK) . te-struct-state . TE(E,tc,struct-state)

  HANDLE(SE,SET-ERROR-STATE,E,tc,es)
  = response-out-TE(SE,ACK) . te-error-state . TE(E,tc,error-state)

  HANDLE(SE,SET-FOCUS-COLOUR,E,tc,es)
  = response-out-TE(SE,ACK) . te-colour-focus . TE(E,tc,es)

  HANDLE(SE,SET-ZONES,E,tc,es)
  = response-out-TE(SE,ACK) . te-register-zones . TE(E,tc,es)

  HANDLE(SE,MOVE-TO-ERROR-POS,E,tc,es)
  = response-out-TE(SE,ACK) . te-cursor-at-error-pos . TE(E,tc,es)

  HANDLE(SE,WRITE-PROTECTED-NEW-FILENAME,E,tc,es)
  = HANDLE(SE,WRITE-PROTECTED-FILENAME,E,tc,es)

  HANDLE(SE,sm,E,tc,es)
  = response-out-TE(SE,ACK) . te-transition . TE(apply(sm,E),tc,es)
end Text-Editor

data module Start-State-Message
begin
  exports
    begin
      functions
        start-state-message-UI : -> STATE-MESSAGE
        start-state-message-TE : -> STATE-MESSAGE
    end
  imports
    State-Message
  equations
  [s1] start-state-message-UI
       = isect(empty-event-set,
               union(add(SE^OPEN-DIALOG-BOX-OTHER-NAME,
                     add(SE^CLOSE-DIALOG-BOX-OTHER-NAME,
                     add(SE^OPEN-BROWSER-NEW-FILENAME,
                     add(SE^CLOSE-BROWSER-NEW-FILENAME,
                     add(USER^WINDOW-RESIZE,
                     add(SE^OPEN-DIALOG-BOX-SAVE-CHANGES,
                     add(SE^CLOSE-DIALOG-BOX-SAVE-CHANGES,
```

```
                     add(USER^QUIT, add(USER^EXPAND,
                     add(USER^PARSE, add(SE^ADD-EXPAND-MENU,
                     add(SE^REM-EXPAND-MENU,
                     add(SE^ADD-DISAMBIGUE-MENU,
                     add(SE^REM-DISAMBIGUE-MENU,
                     empty-event-set)))))))))))))))
  [s2] start-state-message-TE
       = isect(empty-event-set,
               union(add(USER^CHANGE-TEXT,
                     add(SE^WRITE-PROTECTED-FILENAME,
                     add(SE^WRITE-PROTECTED-NEW-FILENAME,
                     add(SE^SAVE-FILENAME, add(SE^CHANGE,
                     add(UI^WINDOW-RESIZE, add(SE^BUFFER-STATE,
                     add(SE^GET-FOCUS-TEXT, add(SE^GET-BUFFER-INFO,
                     add(SE^GET-LINE-LENGTH-INFO,
                     add(SE^GET-TOTAL-LENGTH-INFO, add(SE^CUT,
                     add(SE^PASTE, add(SE^SET-STRUCT-STATE,
                     add(SE^SET-ERROR-STATE, add(SE^SET-ZONES,
                     add(SE^SET-FOCUS-COLOUR,
                     add(SE^MOVE-TO-ERROR-POS,
                     empty-event-set)))))))))))))))))))
end Start-State-Message

process module Structure-Editor
begin
  exports
    begin
      atoms
        event-in-SE     : PARTICIPANT # MESSAGE
        response-in-SE  : PARTICIPANT # MESSAGE
        event-out-SE    : PARTICIPANT # MESSAGE
        response-out-SE : PARTICIPANT # MESSAGE
        event-out-SE    : PARTICIPANT # STATE-MESSAGE
        response-out-SE : PARTICIPANT # STATE-MESSAGE
        se-calculate-menu-items
        se-update-focus-pointer
        se-start-update-tree
        se-end-update-tree
        se-parse-to-struct
        se-parse-to-error
        se-ambiguous-parse
        se-terminate
      processes
        SE
    end
  imports
    Start-State-Message
  processes
    SE                 : EVENT-SET
    HANDLE             : PARTICIPANT # MESSAGE # EVENT-SET
    Yes-change-answer  : EVENT-SET
    Stop               : EVENT-SET
    No-save            : EVENT-SET
    Cancel             : EVENT-SET
    Save               : EVENT-SET
    Write-protected    : EVENT-SET
    Expand             : EVENT-SET
    Item-chosen        : EVENT-SET
    Parse              : EVENT-SET
    Ambiguity          : EVENT-SET
    Struct             : EVENT-SET
    Error              : EVENT-SET
    End-parse          : EVENT-SET
    Close-dialog-box-save-changes
    Close-dialog-box-other-name
    Close-browser-new-filename
    Update-tree
  variables
    E, V, V' : -> EVENT-SET
  definitions
    SE
    = ( event-out-SE(UI,start-state-message-UI) . response-in-SE(UI,ACK) .
        event-out-SE(TE,start-state-message-TE) . response-in-SE(TE,ACK)
      +
      event-out-SE(TE,start-state-message-TE) . response-in-SE(TE,ACK) .
      event-out-SE(UI,start-state-message-UI) . response-in-SE(UI,ACK)
    ) .
    SE(add(UI^QUIT, add(UI^PARSE, add(UI^EXPAND, empty-event-set))))

SE(E)
= sum(p in PARTICIPANT, sum(m in MESSAGE,
      [is-in(p^m,E) = true] -> event-in-SE(p,m) . HANDLE(p,m,E)))

-- QUIT Interaction

HANDLE(UI, QUIT, E)                                                 -- Quit 1
= response-out-SE(UI,rem(USER^EXPAND,rem(USER^QUIT,rem(USER^PARSE)))) .
  event-out-SE(TE,rem(USER^CHANGE-TEXT)) .
  response-in-SE(TE,ACK) .
  event-out-SE(TE,CHANGE) .
  ( response-in-SE(TE,YES) . Yes-change-answer(E)
    +
    response-in-SE(TE,NO) . Stop(E)                                 -- Quit 3
  )

Stop(E)                                                             -- Quit 13
= event-out-SE(UI,isect(empty-event-set,add(SE^TERMINATE))) .
  response-in-SE(UI,ACK) .
  event-out-SE(TE,isect(empty-event-set,add(SE^TERMINATE))) .
  response-in-SE(TE,ACK) .
  ( event-out-SE(UI,TERMINATE) . response-in-SE(UI,ACK)
    ||
    event-out-SE(TE,TERMINATE) . response-in-SE(TE,ACK)
  ) .
  se-terminate

Yes-change-answer(E)                                                -- Quit 2
= event-out-SE(UI,OPEN-DIALOG-BOX-SAVE-CHANGES) .
  response-in-SE(UI,ACK) .
  event-out-SE(UI,add(USER^OK,add(USER^NO,add(USER^CANCEL)))) .
  response-in-SE(UI,ACK) .
  ( event-in-SE(UI,OK) .                                            -- Quit 4
    Close-dialog-box-save-changes .
    Save(E)
    +
    event-in-SE(UI,CANCEL) . Cancel(E)                              -- Quit 5
    +
    event-in-SE(UI,NO) . No-save(E)                                 -- Quit 6
  )

Save(E)
= event-out-SE(TE,WRITE-PROTECTED-FILENAME) .                       -- Quit 4
  ( response-in-SE(TE,YES) . Write-protected(E)                     -- Quit 7
    +
    response-in-SE(TE,NO) .                                         -- Quit 8
    event-out-SE(TE,SAVE-FILENAME) .
    response-in-SE(TE,ACK) .
    Stop(E)
  )

No-save(E)                                                          -- Quit 6
= Close-dialog-box-save-changes .
  Stop(E)

Cancel(E)                                                           -- Quit 5
= Close-dialog-box-save-changes .
  event-out-SE(TE,add(USER^CHANGE-TEXT)) .
  response-in-SE(TE,ACK) .
  event-out-SE(UI,add(USER^EXPAND,add(USER^QUIT,add(USER^PARSE)))) .
  response-in-SE(UI,ACK) .
  SE(E)

Write-protected(E)                                                  -- Quit 7
= event-out-SE(UI,OPEN-DIALOG-BOX-OTHER-NAME) .
  response-in-SE(UI,ACK) .
  event-out-SE(UI,add(USER^OK,add(USER^CANCEL))) .
```

```
  response-in-SE(UI,ACK) .
  (  event-in-SE(UI,OK) .                                         -- Quit 9
     Close-dialog-box-other-name .
     event-out-SE(UI,OPEN-BROWSER-NEW-FILENAME) .
     response-in-SE(UI,ACK) .
     event-out-SE(UI,add(USER^NEW-FILENAME,add(USER^CANCEL))) .
     response-in-SE(UI,ACK) .
     (  event-in-SE(UI,NEW-FILENAME) .                            -- Quit 11
        Close-browser-new-filename .
        Save(E)                                                   -- Quit 7/8
        +
        event-in-SE(UI,CANCEL) .                                  -- Quit 12
        Close-browser-new-filename .
        Stop(E)
     )
     +
     event-in-SE(UI,CANCEL) .                                     -- Quit 10
     Close-dialog-box-other-name .
     Stop(E)
  )

Close-dialog-box-save-changes
= response-out-SE(UI,rem(USER^OK,rem(USER^NO,rem(USER^CANCEL)))) .
  event-out-SE(UI,CLOSE-DIALOG-BOX-SAVE-CHANGES) . response-in-SE(UI,ACK)

Close-dialog-box-other-name
= response-out-SE(UI,rem(USER^OK,rem(USER^CANCEL))) .
  event-out-SE(UI,CLOSE-DIALOG-BOX-OTHER-NAME) . response-in-SE(UI,ACK)

Close-browser-new-filename
= response-out-SE(UI,rem(USER^NEW-FILENAME,rem(USER^CANCEL))) .
  event-out-SE(UI,CLOSE-BROWSER-NEW-FILENAME) . response-in-SE(UI,ACK)

-- PARSE Interaction

HANDLE(UI, PARSE, E)                                             -- Parse 1
= response-out-SE(UI,rem(USER^EXPAND,rem(USER^QUIT,rem(USER^PARSE)))) .
  event-out-SE(TE,rem(USER^CHANGE-TEXT)) .
  response-in-SE(TE,ACK) .
  event-out-SE(TE,BUFFER-STATE) .
  ( response-in-SE(TE,TEXT) .                                    -- Parse 2
    event-out-SE(TE,GET-BUFFER-INFO) .
    response-in-SE(TE,BUFFER-INFO) .
    event-out-SE(TE,GET-FOCUS-TEXT) .
    response-in-SE(TE,FOCUS-TEXT) .
    se-update-focus-pointer .
    Parse(E)                                               -- Parse 5, 6, 7
    +
    response-in-SE(TE,STRUCT) . End-parse(E)                     -- Parse 3
    +
    response-in-SE(TE,ERROR) . End-parse(E)                      -- Parse 4
  )

Parse(E)
= ( se-ambiguous-parse . Ambiguity(E)                            -- Parse 5
    +
    se-parse-to-struct . Struct(E)                               -- Parse 6
    +
    se-parse-to-error . Error(E)                                 -- Parse 7
  )

Ambiguity(E)                                                     -- Parse 5
= event-out-SE(UI,ADD-DISAMBIGUE-MENU) .
  response-in-SE(UI,ACK) .
  event-out-SE(UI,add(USER^CHOICE1,add(USER^CHOICE2,add(USER^CANCEL)))) .
  response-in-SE(UI,ACK) .
  (  event-in-SE(UI,CHOICE1) .                                   -- Parse 8
     response-out-SE(UI,rem(USER^CHOICE1,
                       rem(USER^CHOICE2,rem(USER^CANCEL)))) .
     event-out-SE(UI,REM-DISAMBIGUE-MENU) .
     response-in-SE(UI,ACK) .
     Parse(E)
     +
     event-in-SE(UI,CHOICE2) .                                   -- Parse 8
     response-out-SE(UI,rem(USER^CHOICE1,
                       rem(USER^CHOICE2,rem(USER^CANCEL)))) .
     event-out-SE(UI,REM-DISAMBIGUE-MENU) .
     response-in-SE(UI,ACK) .
     Parse(E)
     +
     event-in-SE(UI,CANCEL) .                                    -- Parse 9
     response-out-SE(UI,rem(USER^CHOICE1,
                       rem(USER^CHOICE2,rem(USER^CANCEL)))) .
     event-out-SE(UI,REM-DISAMBIGUE-MENU) .
     response-in-SE(UI,ACK) .
     End-parse(E)
  )

Struct(E)                                                        -- Parse 6
= event-out-SE(TE,SET-STRUCT-STATE) .
  response-in-SE(TE,ACK) .
  Update-tree .
  event-out-SE(TE,SET-ZONES) .
  response-in-SE(TE,ACK) .
  event-out-SE(TE,SET-FOCUS-COLOUR) .
  response-in-SE(TE,ACK) .
  End-parse(E)

Error(E)                                                         -- Parse 7
= event-out-SE(TE,SET-ERROR-STATE) .
  response-in-SE(TE,ACK) .
  event-out-SE(TE,MOVE-TO-ERROR-POS) .
  response-in-SE(TE,ACK) .
  event-out-SE(TE,SET-FOCUS-COLOUR) .
  response-in-SE(TE,ACK) .
  End-parse(E)

End-parse(E)                                                     -- Parse 10
= event-out-SE(TE,add(USER^CHANGE-TEXT)) .
  response-in-SE(TE,ACK) .
  event-out-SE(UI,add(USER^EXPAND,add(USER^QUIT,add(USER^PARSE)))) .
  response-in-SE(UI,ACK) .
  SE(E)

Update-tree
= se-start-update-tree .
  event-out-SE(TE,GET-LINE-LENGTH-INFO) .
  response-in-SE(TE,LINE-LENGTH-INFO) .
  event-out-SE(TE,GET-TOTAL-LENGTH-INFO) .
  response-in-SE(TE,TOTAL-LENGTH-INFO) .
  se-end-update-tree

-- EXPAND Interaction

HANDLE(UI, EXPAND, E)                                           -- Expand 1
= response-out-SE(UI,rem(USER^EXPAND,rem(USER^QUIT,rem(USER^PARSE)))) .
  event-out-SE(TE,rem(USER^CHANGE-TEXT)) .
  response-in-SE(TE,ACK) .
  event-out-SE(TE,BUFFER-STATE) .
  ( response-in-SE(TE,TEXT) .                                   -- Expand 2
    event-out-SE(TE,GET-BUFFER-INFO) .
    response-in-SE(TE,BUFFER-INFO) .
    se-update-focus-pointer
    +
    response-in-SE(TE,STRUCT)                                   -- Expand 3
    +
    response-in-SE(TE,ERROR)                                    -- Expand 4
  ) .
  Expand(E)                                                     -- Expand 5

Expand(E)                                                       -- Expand 5
= se-calculate-menu-items .
  event-out-SE(UI,ADD-EXPAND-MENU) .
  response-in-SE(UI,ACK) .
  event-out-SE(UI,add(USER^CHOICE1,add(USER^CHOICE2,add(USER^CANCEL)))) .
  response-in-SE(UI,ACK) .
```

```
      (  event-in-SE(UI,CHOICE1) . Item-chosen(E)                  -- Expand 6
         +
         event-in-SE(UI,CHOICE2) . Item-chosen(E)                  -- Expand 6
         +
         event-in-SE(UI,CANCEL) .                                  -- Expand 7
         response-out-SE(UI,rem(USER^CHOICE1,
                                rem(USER^CHOICE2,rem(USER^CANCEL)))) .
         event-out-SE(UI,REM-EXPAND-MENU) .
         response-in-SE(UI,ACK)
      ) .
      event-out-SE(TE,add(USER^CHANGE-TEXT)) .                     -- Expand 8
      response-in-SE(TE,ACK) .
      event-out-SE(UI,add(USER^EXPAND,add(USER^QUIT,add(USER^PARSE)))) .
      response-in-SE(UI,ACK) .
      SE(E)

    Item-chosen(E)                                                 -- Expand 6
    = response-out-SE(UI,rem(USER^CHOICE1,rem(USER^CHOICE2,rem(USER^CANCEL)))) .
      event-out-SE(UI,REM-EXPAND-MENU) .
      response-in-SE(UI,ACK) .
      Update-tree     .
      event-out-SE(TE,CUT)   .
      response-in-SE(TE,ACK) .
      event-out-SE(TE,PASTE) .
      response-in-SE(TE,ACK) .
      event-out-SE(TE,SET-STRUCT-STATE) .
      response-in-SE(TE,ACK) .
      event-out-SE(TE,SET-FOCUS-COLOUR) .
      response-in-SE(TE,ACK)
end Structure-Editor

process module Generic-Structure-Editor
begin
  exports
    begin
      atoms
        event    : PARTICIPANT # PARTICIPANT # MESSAGE
        response : PARTICIPANT # PARTICIPANT # MESSAGE
        event    : PARTICIPANT # PARTICIPANT # STATE-MESSAGE
        response : PARTICIPANT # PARTICIPANT # STATE-MESSAGE
      processes
        GSE
    end
  imports
    User-Interface, Text-Editor, Structure-Editor
  sets
    of PARTICIPANT UI-SE-TE = {UI, SE, TE}
    of atoms
    H = { event-in-TE(p,m),  event-in-UI(p,m) | p in UI-SE-TE, m in MESSAGE }
        +
        { event-out-TE(p,m)  , event-out-UI(p,m)   ,
          event-in-SE(p,m)   , event-out-SE(p,m)   ,
          response-in-TE(p,m), response-out-TE(p,m),
          response-in-SE(p,m), response-out-SE(p,m),
          response-in-UI(p,m), response-out-UI(p,m)
          | p in PARTICIPANT, m in MESSAGE }
        +
        { event-in-TE(p,m)    , event-out-SE(p,m)  ,
          event-in-UI(p,m)    , response-in-TE(p,m),
          response-out-SE(p,m), response-in-UI(p,m)
          | p in PARTICIPANT, m in STATE-MESSAGE }

  communications
    event-in-TE(UI,m) | event-out-UI(TE,m)       = event(UI,TE,m)
      for m in MESSAGE
    response-in-TE(UI,m) | response-out-UI(TE,m) = response(UI,TE,m)
      for m in MESSAGE
    event-in-UI(TE,m) | event-out-TE(UI,m)       = event(TE,UI,m)
      for m in MESSAGE
    response-in-UI(TE,m) | response-out-TE(UI,m) = response(TE,UI,m)
      for m in MESSAGE

    event-in-TE(SE,m) | event-out-SE(TE,m)       = event(SE,TE,m)
      for m in MESSAGE
    response-in-TE(SE,m) | response-out-SE(TE,m) = response(SE,TE,m)
      for m in MESSAGE
    event-in-SE(TE,m) | event-out-TE(SE,m)       = event(TE,SE,m)
      for m in MESSAGE
    response-in-SE(TE,m) | response-out-TE(SE,m) = response(TE,SE,m)
      for m in MESSAGE

    event-in-UI(SE,m) | event-out-SE(UI,m)       = event(SE,UI,m)
      for m in MESSAGE
    response-in-UI(SE,m) | response-out-SE(UI,m) = response(SE,UI,m)
      for m in MESSAGE
    event-in-SE(UI,m) | event-out-UI(SE,m)       = event(UI,SE,m)
      for m in MESSAGE
    response-in-SE(UI,m) | response-out-UI(SE,m) = response(UI,SE,m)
      for m in MESSAGE

    event-in-UI(SE,m) | event-out-SE(UI,m)       = event(SE,UI,m)
      for m in STATE-MESSAGE
    response-in-UI(SE,m) | response-out-SE(UI,m) = response(SE,UI,m)
      for m in STATE-MESSAGE

    event-in-TE(SE,m) | event-out-SE(TE,m)       = event(SE,TE,m)
      for m in STATE-MESSAGE
    response-in-TE(SE,m) | response-out-SE(TE,m) = response(SE,TE,m)
      for m in STATE-MESSAGE

  definitions
    GSE = encaps(H, UI || TE || SE)
end Generic-Structure-Editor
```