Appendix A

The Noos development environment

This Appendix describes the Noos development environment. The Noos language is implemented using Common Lisp [Steele, 1990] and currently is running in several platforms. The main development platform is the MacOS (using MCL [Digitool, 1996]), but it is also available for Unix machines and PCs (both using Clisp[1]).

The purpose of this Appendix is to show some of the tools implemented in the Noos development environment for assisting the modeling and constructing of applications in Noos. Our purpose is not to provide a user manual for developing applications in Noos.

A.1 Defining feature terms in Noos

There are two ways of adding new terms in Noos: by defining descriptions in a file and then loading the file into the Noos environment[2] or by directly typing descriptions on the Noos listener. For instance, if we type the `person`'s description on the listener (as shown in Figure A.1) the returned value is `<Person>`, which is the "print-name" of the `person` feature term.

Lazy evaluation

Lazy (on demand) evaluation means that no expression is evaluated unless it is needed for some computation. A description can thus refer to the name of another description even if it is defined later—on the listener or on the file being loaded or on another file. However, refinement appearing on the root of a description do require the constituent to be already defined. The reason is that

[1]Clisp is a public domain Lisp available at <ftp://ma2s2.mathematik.uni-karlsruhe.de/pub/lisp/clisp>.

[2]Files can be loaded by using the `load` Lisp function in the Listener.

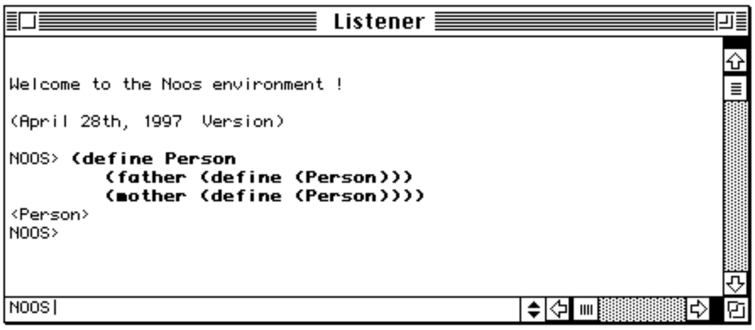

Figure A.1. Defining a new feature term on the Noos listener.

the outmost `define` in a description (the root) *is* evaluated when loading a file or typing on the listener. Anonymous descriptions appearing inside a description are not evaluated until needed. For this reason subdescriptions can refine a description that is defined later—or the same description being defined (see Figure A.1 where `father` and `mother` features of `person` are defined recursively by refinement of `person`).

Compact descriptions

In order to provide a more compact notation for the definition of closed methods in features, the syntax of Noos is extended. Using this extended syntax, a closed method defined by refinement of a built-in method can be defined using a position-based notation for specifying all of its required feature values.

The syntax used for specifying compact descriptions is the following:

(*built-in-name* $\text{param}_1 \cdots \text{param}_n$)

where *built-in-name* is the name of a built-in method, and each $\text{param}_1, \ldots, \text{param}_n$ is a feature value, a path reference, or a compact description. Compact descriptions only can be used for specifying feature values.

The order of the required features is fixed by Noos. Appendix C describes the names of the built-in methods that accept compact descriptions and the order of specification of their required features.

For instance, the compact notation for the `identity?`, the `conditional`, and the `lower-than?` built-in methods is the following:

(`identity?` item1 item2)

(`if` condition result otherwise)

(`<` is-lower than)

where the names of the parameters indicate the specification order for the required features of each built-in method.

Using the compact notation, the following two closed methods defined in features `gas-gauge-reading` and `empty-level?`:

```
(define Bills-car
  (owner Bill)
  (gas-level-in-tank 2)
  ((gas-gauge-reading (define (conditional)
                        ((condition (define (lower-than?)
                                       (is-lower (>> gas-level-in-tank))
                                       (than 5))))
                        (result empty)
                        (otherwise full))))
  ((empty-level?  (define (Identity?)
                     (item1 empty)
                     (item2 (>> gas-gauge-reading))))))
```

can be equivalently specified using the compact notation as follows:

```
(define Bills-car
  (owner Bill)
  (gas-level-in-tank 2)
  (gas-gauge-reading (if (< (>> gas-level-in-tank) 5)
                         empty
                         full))
  (empty-level?  (Identity?  empty (>> gas-gauge-reading))))
```

Note that the use of this compact notation requires that all required features have to be specified and that no other features can be specified. A second remark is that compact descriptions are specified using a single parenthesis in the same way as path references. Another remark is that using the compact notation we can only use compact descriptions for defining methods in the features—using the compact notation we can only define feature values using either constant values, path references, or compact descriptions. Moreover, since the name of the features are not specified, the position of the parameters determines the feature that a parameter is referred to.

The use of the compact notation allows also to define compositions of the same built-in method—such as arithmetic methods and methods for manipulating sets—in a easy way. For instance, since the `addition` built-in method has two required features, the sum of three numbers `a`, `b`, and `c`, using the compact syntax, has to be specified as the following composition:

```
(+ a (+ b c))
```

This compact syntax is extended for defining compositions as follows:

```
(+ a b c)
```

For instance, we can define the `earnings` of a specific company `comp` as the sum of the earnings of its three `production-lines` as follows:

```
(define (Company Comp)
  (line-1 (define (production-line)
            (line-name cosmetics)
            (earnings 12000)))
  (line-2 (define (production-line)
            (line-name toys)
            (earnings 23000)))
  (line-3 (define (production-line)
            (line-name nourishment)
            (earnings 8000)))
  (earnings (+ (>> earnings line-1)
               (>> earnings line-2)
               (>> earnings line-3))))
```

Multiple path references

Another language extension is to allow specifications of multiple path references in a feature. These multiple path references are interpreted using the union built-in method. In fact, the following description:

```
(define Person
  (uncles (>> brothers of mother)
          (>> brother of father)))
```

is a short syntax equivalent to the following description:

```
(define Person
  ((uncles (define (union)
             (item1 (>> brothers of mother))
             (item2 (>> brother of father)))))
```

Engaging inference

When a user types a path reference or an eval expression in the Listener, it is interpreted as a *query-expression* to be answered by the system. The Listener, however, does not accept relative path references. The reason is that a relative path reference only can be bound in the scope of a description. An alternative way to engage the inference is directly using the pop-up menus provided in the browsers (see Section A.4).

We have seen in Section 3.3.5 that, since reflective operations are references to feature terms, path references and reflective operations can be combined. These combined expressions are extended query-expressions that can be posed to the system in the listener. For instance, the following query-expression:

```
(meta (>> father of Tom))
```

asks for the metalevel of the person that is the `father` of `Tom`.

Whenever Noos yields a solution value for a problem task, the user can force backtracking to the Noos inference engine to search for another solution value for the task using the `force-backtracking` command.

For instance, let us assume that the following two descriptions have been added to Noos:

```
(define (person professional)
  ((phone-number (reify (>> phone-number spouse))
                 (reify (>> phone-number home))
                 (reify (>> phone-number works-in)))))

(define (professional Carol)
  (spouse (define (person)
            (phone-number 3344)))
  (works-in (define (company)
              (phone-number 8766))))
```

Then, we can pose the following query-expression to the listener:

```
(>> phone-number of Carol)
```

The answer yielded by Noos will be `3344`. Next, forcing the backtracking with the `force-backtracking` command as follows:

```
(force-backtracking)
```

the inference is resumed in Noos yielding `8766`. If we force backtracking once more, the value yielded by Noos will be `Fail`—the token that Noos yields for indicating that no more values can be inferred.

A.2 The predefined sort hierarchy of Noos

Noos provides an initial set of sorts with an order relation among them. There is a top sort called `any`. `Any` represents the minimum information and all the other sorts are more specific than `any` (for each sort `S` we have that `any` $\leq$ `S`). Predefined sorts are (see Figure A.2):

- all numbers and the sort `number`, with the order relations `number` $\leq n$ for all numbers n,
- all strings and the sort `string`, with the order relations `string` $\leq s$ for all strings s,
- all symbols and the sort `symbol`, with the order relations `symbol` $\leq s$ for all symbols s,
- sorts `boolean`, `true`, and `false` with the order relations `boolean` $\leq$ `true` and `boolean` $\leq$ `false`,
- sorts `set` and `empty-set` with the order relation `set` $\leq$ `empty-set`,
- all built-in methods and the sort `method`, with the order relations `method` $\leq m$ for all built-in methods m,

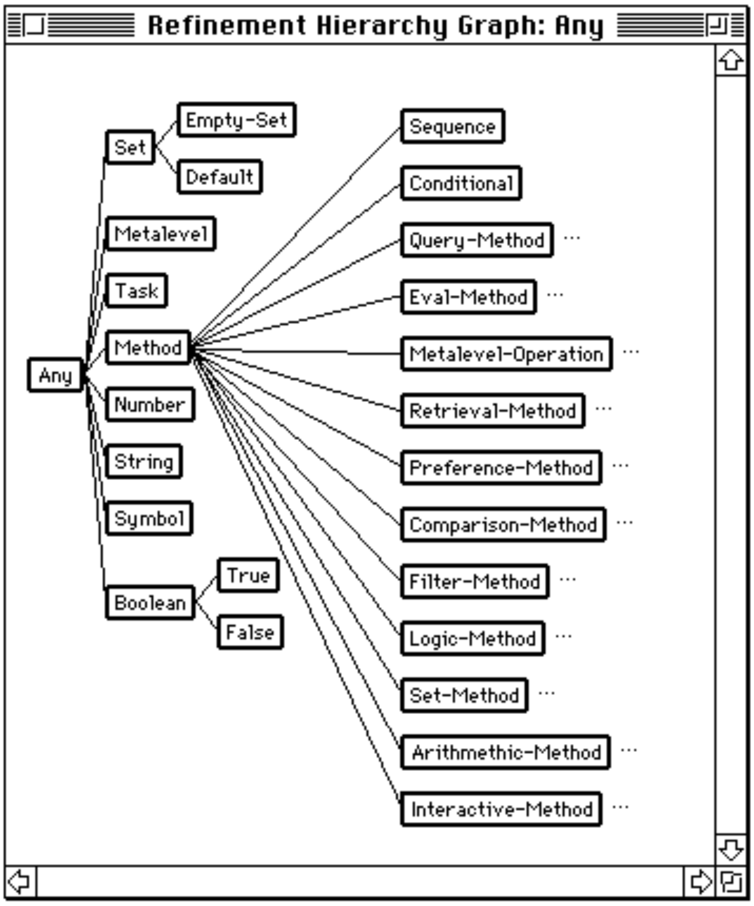

Figure A.2. The predefined sort hierarchy of Noos.

- sorts `metalevel`, `default`, and `task`.

The complete set of predefined built-in methods is described in Appendix D.

From this set of initial sorts new sorts can be defined, using refinement, for specifying the sort hierarchy for a given domain.

A.3 Episodic memory

Whenever a problem task is solved by Noos, the user can decide to incorporate the episodic model built to the Noos episodic memory using the `freeze` command. For instance, after solving the `phone-number` of `Carol` we can type the following command:

```
(freeze Carol)
```

that will incorporate the episodic model built in solving `phone-number(Carol)` problem task into the episodic memory.

When an episodic model is incorporated to the episodic memory, no more inference can be engaged on this episodic model.

After solving a problem task, the user can decide not to incorporate the episodic model built in solving the task. For this situation, the Noos environment provides the `forget!` command. Specifying the `forget!` command over a feature term, the term is removed and will become inaccessible—and consequently no more inference can be engaged on this term.

Noos provides a way to explicitly determine that a specific feature term has not to be incorporated to the episodic model using the `:ephemeral` token as following:

`(define (`*constituent name* `) :ephemeral` body`)`

Specifically, the `freeze` command incorporates into the episodic memory all the feature terms belonging to an episodic model, excepting those explicitely specified as `:ephemeral`. For instance, the `:ephemeral` token is frequently used in retrieval patterns because a retrieval pattern is a term that is only constructed for retrieval purposes and it should not be retrieved in next problems.

Customizing Episodic memory

Since there are applications developed in Noos that will not require all the capabilities provided by the episodic memory of Noos, episodic memory can be customized for increasing the efficiency of the system.

A first option is to deactivate the episodic memory when the application does not use it. For instance, the SHAM application described in Section 6.3 solves problem tasks only using problem solving methods modeling background musical knowledge. Thus, episodic memory capabilities are deactivated in SHAM since they are not used.

Episodic memory can be deactivated using the `exclude` command with the `:all` token as follows:

```
(exclude :all)
```

Another option is to specify that some kind of terms are not to be incorporated into the episodic memory. For instance, there are applications that only require the retrieval of feature values and do not reason about the methods used in a specific task. These applications (such as *Saxex* CHROMA, SPIN, and GYMEL) do not require to introspect tasks, methods, and metalevels. The `exclude` command can also be used for this purpose, as follows:

```
(exclude task method metalevel)
```

The `exclude` command accepts any combination of some of the three different categories specified in the example: `task`, `method`, and `metalevel`. For instance, another legal option is the following:

```
(exclude task metalevel)
```

Finally, a third option is to specify that methods used in solving tasks can be removed after their evaluation. This option requires methods to be excluded to the episodic memory. The advantage of this option is that the Noos memory required by an application is lower. The disadvantage is that neither reflective operations nor browsers over the methods can be applied. The `compact-methods` command can be used for activate and deactivate this option using the following syntax respectively:

```
(compact-methods t)
(compact-methods nil)
```

A.4 Browsing

The Noos development environment provides four types of graphical browsing facilities: a *feature term browser*, a *task/method decomposition browser*, a *task structure browser*, and a *refinement hierarchy browser*.

There are two possibilities for browsing: using a text-based browser that can display into the listener or into a file, or using a window-based graphical interface. The first option is available for all the implementations of Noos. The window-based browsers are only available on the MCL implementation.

Text-based browsers and window-based browsers can be displayed typing specific browsing commands on the listener. Window-based browsers can be also displayed using the Noos menu developed for the MCL version.

A.4.1 Feature term browser

The feature term browser provides a graphical representation of a feature term. The `browse` command can be used to start a feature term browser using the following syntax:

(`browse` *name* [*depth*])

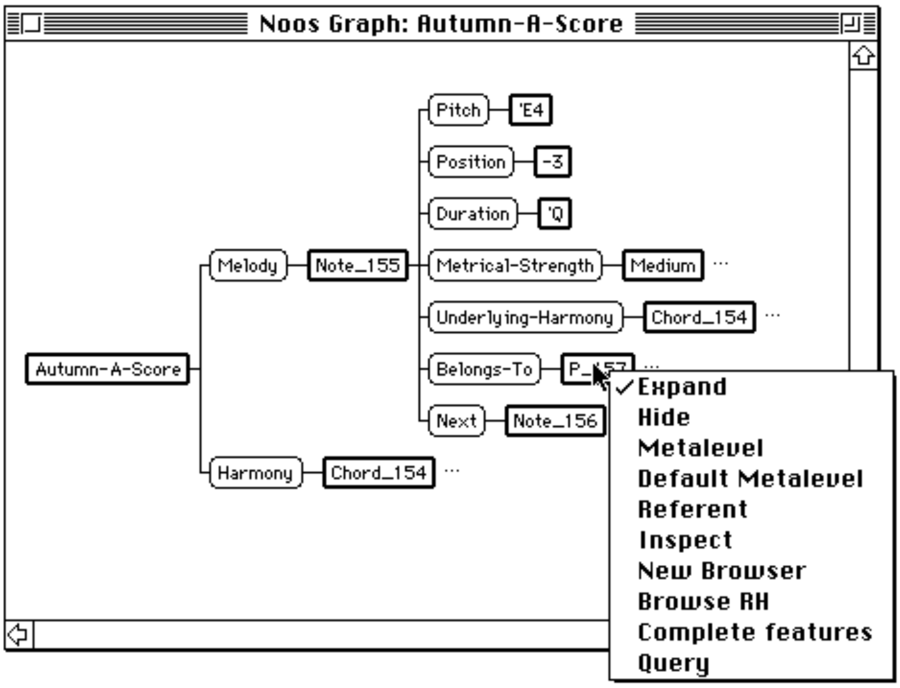

Figure A.3. A browser of the score of 'Autumn Leaves' ballad from *Saxex* application. A pop-up menu has been activated by clicking the `P_157` node.

where *name* is an identifier of a feature term and *depth* is an optional integer number indicating the depth level of the term to be displayed. The default value for the depth level for all the Noos browsers is 3.

For instance, a browser of the first phrase of the 'Autumn Leaves' ballad from *Saxex* application (described in Section 6.5) can be started by the following command:

```
(browse Autumn-A-Score)
```

obtaining the browser window of Figure A.3. Note that terms are displayed as graphs with nodes representing their identifiers as thick boxes; features are represented as thin boxes; ellipsis (three dots) indicate nodes amenable to be expanded; and gray boxes express references to existing nodes—nodes expanded in another place in the window.

Using the default value for the depth parameter (3), a feature term browser displays a root node (depth 1) and their features: the feature names (depth 2) and the feature values (depth 3). All feature values amenable to be expanded are indicated with ellipsis.

Many browsers can be open at the same time. Each browser is displayed in a different window. Browsers can be dynamically expanded using *pop-up menus*. Specifically, there are two kinds of pop-up menus: one for nodes and another for features (in Figure A.3 a pop-up menu for terms is shown).

The pop-up menu unfolded when the user clicks on a node box has the following ten commands:

- *Expand*: the *expand* command allows to expand a node's term displaying its features. The features are displayed without engaging the inference of their values (i.e. only computed and constant values are shown). When a feature value is neither a constant value nor it has been inferred, a node with label `Unknown` is displayed. The expand command performed on this unknown node will engage the inference of that feature value. The result yielded is a node with the value, if it can be inferred, or, otherwise, a node with the `FAIL` label.

- *Hide*: the *hide* command is the converse command to expand: the features of a node are hidden and ellipsis are displayed for indicating that the node is amenable to be expanded.

- *Metalevel*: the *metalevel* command starts a new browser displaying the metalevel term of the selected node. When the metalevel has not been defined, no new browser is started.

- *Default metalevel*: the *default metalevel* command starts a new browser displaying the default metalevel term of the selected node. When the default metalevel is not defined, no new browser is started.

- *Referent*: the *referent* command is the converse command to the *metalevel* command. The *referent* command starts a new browser displaying the referent term of the selected node. When the referent does not exist, no new browser is started.

- *Inspect*: the *inspect* command starts the inspector of Lisp. We have extended the Lisp inspector for displaying Noos terms.

- *New browser*: the *new browser* command starts a new browser displaying the selected node. This command is not allowed over the root node. If there is another browser where the selected node is the root, this browser window is activated and no new browser is started.

- *Browse RH*: the *browse RH* command starts a refinement hierarchy browser with the sort of the selected node as root (see Section A.4.4).

- *Complete features*: Following the lazy approach of inference in Noos, in definitions by refinement, features defined in the constituent are only incorporated to the refined term when they are needed for any computation. This means that when a term is expanded in a browser only features defined in the term and features inferred in some problem task will be displayed. The *complete features* command forces to display all the features defined. This option only displays features without engaging the inference for their values. Next, the user can use the *expand* command for engaging the inference in a specific feature.

- *Query*: finally, the query command on a node allows to engage the inference on any feature for that node by means of specifying a feature name.

The pop-up menu unfolded when the user clicks on a feature box has the following six options:

- *Expand*: the *expand* command, analogously to the previous pop-up menu, allows to expand a feature displaying its value. The value is displayed without engaging inference. When the feature value is neither a constant value nor it has been inferred, a node with label `Unknown` is displayed.
- *Hide*: the *hide* command is the converse command to expand: the feature value of a feature is hidden.
- *Task*: the *task* command starts a new browser displaying the task term of the selected feature.
- *Method*: the *method* command starts a new browser displaying the method term defined for the selected feature. When no method is defined, no new browser is started.
- *Task structure*: the *task structure* command starts a task structure browser displaying the episodic model constructed for inferring the value of the feature (see Section A.4.3).
- *Task decomposition*: the *task decomposition* command starts a browser displaying the task/method decomposition defined for the selected feature (see Section A.4.2).

The text-based version of a feature term browser can be obtained using the `tbrowse` command. For instance, the score shown in Figure A.3 yielded by the following command:

```
(tbrowse Autumn-A-Score 5)
```

obtaining the text-based browser presented in Figure A.4.

A.4.2 Task/method decomposition browser

The task/method decomposition browser provides a graphical representation of the recursive decomposition of a task into subtasks by means of methods. That is to say, this kind of browser provides a graphical representation for the problem solving knowledge (See Section 3.1). The `browse-task` command and the `browse-method` command can be used, respectively, to start a browser for a specific task or a browser for a specific method using the following syntax respectively:

```
{Autumn-A-Score
  (Melody {Note_155
             (Pitch E4)
             (Position -3)
             (Duration Q)
             (Metrical-Strength {Medium})
             (Underlying-Harmony {Chord_154})
             (Belongs-To P_157)
             (Next {Note_156})})
  (Harmony {Chord_154
             (Name Cmaj7)
             (Position 0)
             (Duration W)
             (Next {Chord_155})})}
```

Figure A.4. A text-based feature term browser of the score of 'Autumn Leaves' ballad from *Saxex* application.

(`browse-task` *task-name term-name* [*depth*])

(`browse-method` *method-name* [*depth*])

For instance, a browser of the task/method decomposition of `general-diagnosis` method following [Benjamins, 1993] can be started by the following command:

```
(browse-method general-diagnosis)
```

obtaining the browser window of Figure A.5. Tasks are drawn with thick boxes; methods are drawn with thin boxes; and ellipsis indicate the nodes amenable to be expanded.

Note that a method can be displayed using a feature term browser and using a task/method decomposition browser. Nevertheless, the information displayed in each browser is different. The first one shows the method's (sub)tasks and their values; the task/method decomposition browser shows the method's tasks and their (sub)methods.

Many task/method decomposition browsers can be open at the same time and each browsing is displayed in a different window. Browsers can also be dynamically expanded using *pop-up menus*. Similarly to feature term browsers, there are two kinds of pop-up menus: one pop-up menu for methods and another for tasks.

The set of commands allowed in methods are the same as before, excluding the *query* command: *expand*, *hide*, *metalevel*, *default metalevel*, *referent*, *inspect*, *new browser*, *browse RH*, and *complete features*.

The set of commands allowed in tasks are the following: *expand*, *hide*, *referent*, *inspect*, and *new browser*.

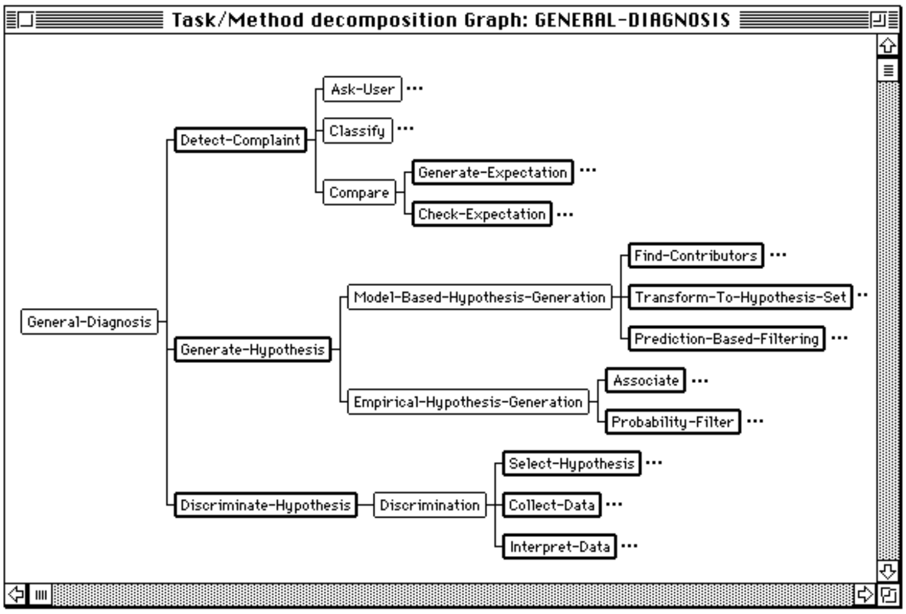

Figure A.5. A browser of the task/method decomposition for the general diagnosis method.

The text-based version of a task/method decomposition browser can be obtained using the `tbrowse-task` and the `tbrowse-method` commands. For instance, the task/method decomposition for the general diagnosis method shown in Figure A.5 yielded by the following command:

```
(tbrowse-method General-Diagnosis 3)
```

will be the following:

```
{General-Diagnosis
  (Detect-Complaint {Ask-User}
                    {Classify}
                    {Compare})
  (Generate-Hypothesis {Model-Based-Hypothesis-Generation}
                       {Empirical-Hypothesis-Generation})
  (Discriminate-Hypothesis {Discrimination})}
```

A.4.3 Task structure browser

The task structure browser provides a graphical representation of the part of the episodic model concerning the methods and subtasks effectively involved in

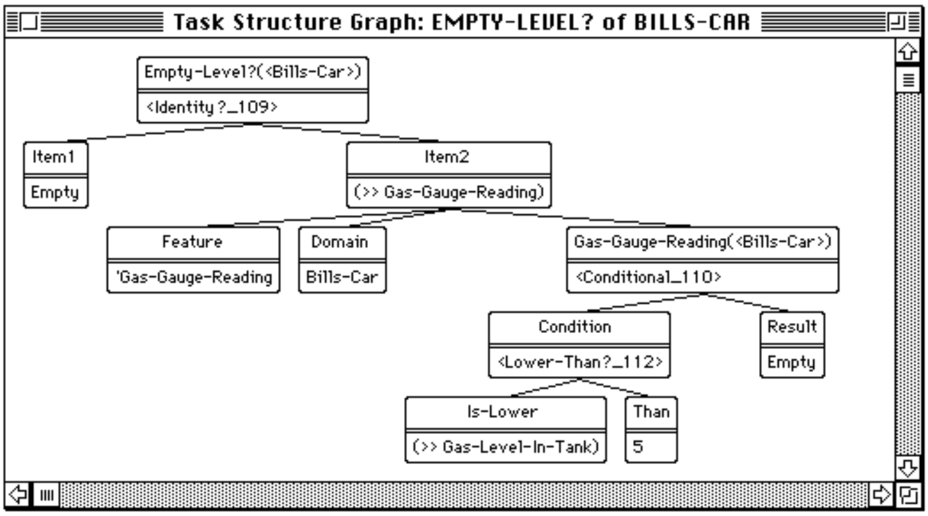

Figure A.6. A task structure browser from the episodic model of problem task `empty-level?(Bills-car)`.

solving a specific problem task. The `browse-stask` command can be used to start a browser for a specific task using the following syntax:

(`browse-stask` *feature-name term-name* [*depth*])

For instance, a task structure browser from the episodic model of problem task `empty-level?(Bills-car)` can be started by the following command:

```
(browse-stask empty-level?  Bills-car)
```

obtaining the browser window of Figure A.6. Each node in the browser shows a task and the method that has solved that task. The upper part shows the task name and the lower part shows the method printname.

Note that only the subtasks effectively involved in solving a specific method are displayed. For instance, the subtask decomposition of the `conditional_110` method of Figure A.6 is only composed of the `condition` task and the `result` task since the result of the `condition` task is `true`.

Another example of a task structure browser built after solving the `cup?(Obj1)` problem task can be found in Section 4.7.

Task structure browsers can also be dynamically expanded using *pop-up menus*. The pop-up menus have the following five commands: *expand*, *hide*, *referent*, *inspect*, and *new browser*.

Note that the referent of a task is the result value inferred by the evaluation of its method (see Section 3.3.5).

The text-based version of a task structure browser can be obtained using the `tbrowse-stask` command. For instance, the task structure from the episodic

```
{Saxex-sort}
   {Saxex-Problem}
   {Score-element}
      {Phrase}
      {Note}
      {Chord}
   {Analysis-Element}
      {Phrase-Analysis}
      {Nstructure}
      {Time-Span-Node}
      {Prol-Reduc-Node}
   {Performance-Element}
      {Phrase-Performance}
      {Event}
      {Label}
```

Figure A.7. A text-based feature refinement hirerarchy browser of the *Saxex* application.

model of problem task `empty-level?(Bills-car)` shown in Figure A.6 yielded by the following command:

```
(tbrowse-stask empty-level?  Bills-car 8)
```

will be the following:

```
(Empty-Level?(Bills-Car)
  {Identity?_109
    (Item1 {Empty})
    (Item2 {Infer-value (>> Gas-Gauge-Reading)
             (Feature 'Gas-Gauge-Reading)
             (Domain {Bills-Car})
             (Gas-Gauge-Reading(Bills-Car)
               {Conditional_110
                 (Condition {Lower-Than_112})
                 (Result {True})})})})
```

A.4.4 Refinement hierarchy browser

Finally, the refinement hierarchy browser provides a graphical representation of the sort hierarchy defined in developing a specific application refining the built-in sort hierarchy of Noos. The `browse-RH` command can be used to start a browser from a specific sort using the following syntax:

(`browse-RH` *sort-name* [*depth*])

For instance, the refinement hierarchy browser defined for the *Saxex* application can be started by the following command:

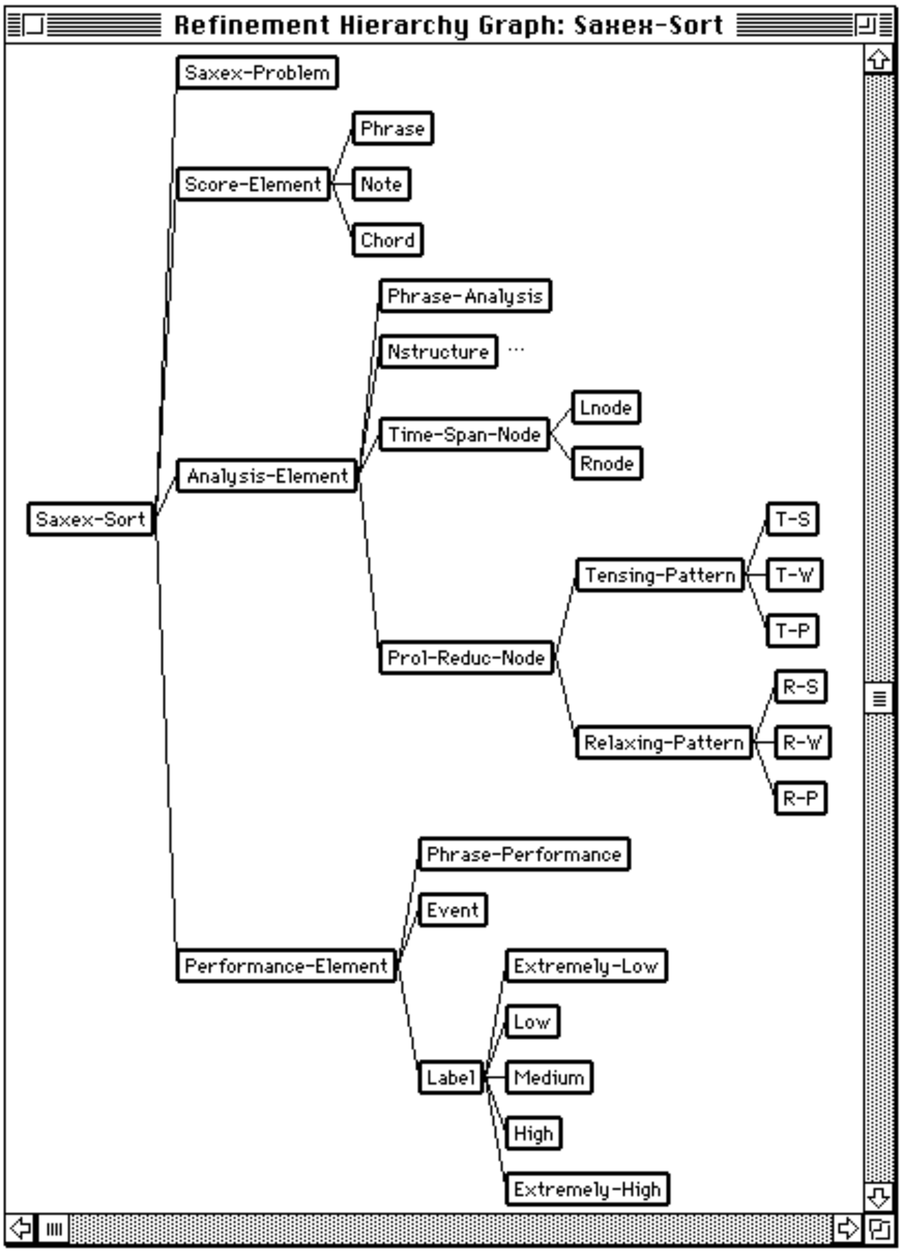

Figure A.8. A refinement hierarchy browser from the *Saxex* application.

```
(browse-RH Saxex-sort)
```

obtaining the browser window of Figure A.8.

This kind of browsers can also be dynamically expanded using *pop-up menus*. The pop-up menus have the following four commands: *expand*, *hide*, *inspect*, and *new browser*.

The text-based version of a refinement hierarchy browser can be obtained using the `tbrowse-rh` command. For instance, the refinement hierarchy browser defined for the *Saxex* application shown in Figure A.8 yielded by the following command:

```
(tbrowse-rh Saxex-sort 3)
```

obtaining the text-based browser presented in Figure A.7.

A.5 Tracing

The Noos development environment provides two commands for tracing the inference: `usual-trace` and `trace-feature`.

The `usual-trace` command enables a complete trace of the inference. The trace can be disabled using the `no-trace` command. Examples of Noos traces have already shown in Section 3.2.5 and in Section 3.5.

Figure A.9 shows the trace generated in solving the `empty-level?(bills-car)` problem task that has the episodic model displayed in Figure A.6.

The `trace-feature` command enables a selective trace on a set of feature names specified by the user. Since the complete trace generated in solving a complex problem task may be too large, the `trace-feature` trace option allows to trace only the representative features.

The `trace-feature` command is disabled using the `disable-trace` command. Both commands `usual-trace` and `trace-feature` can be used simultaneously.

For instance, the inference trace generated in solving problem task `empty-level?(bills-car)` with three features selected for trace as follows:

```
(trace-feature empty-level?  gas-gauge-reading gas-level-in-tank)
```

will be the following trace:

```
(>> empty-level?  of bills-car)

 1 Query: empty-level?(<bills-car>)
 2    Query: gas-gauge-reading(<bills-car>)
 3       Query: gas-level-in-tank(<bills-car>)
 4       Result: 2
 5    Result: <empty>
 6 Result: <true>

<true>
```

```
(>> empty-level?  of bills-car)

Eval: <Infer-value (>> empty-level?  of bills-car)>
⟹
   Task: empty-level?(<bills-car>)
   Eval: <identity?_109>
   ⟹
     Task: item1(<identity?_109>)
     Value: <true>
     Task: item2(<identity?_109>)
     Eval: <Infer-value (>> gas-gauge-reading)>
     ⟹
        Task: feature(<Infer-value (>> gas-gauge-reading)>)
        Value: 'gas-gauge-reading
        Task: domain(<Infer-value (>> gas-gauge-reading)>)
        Value: <bills-car>
        Task: gas-gauge-reading(<bills-car>)
        Eval: <conditional_110>
        ⟹
          Task: condition(<conditional_110>)
          Eval: <lower-than?_112>
          ⟹
             Task: is-lower(<lower-than?_112>)
             Eval: <Infer-value (>> gas-level-in-tank)>
             Value: 2
             Task: than(<lower-than?_112>)
             Value: 5
          ⟸
          Value: <true>
          Task: result(<conditional_110>)
          Value: <empty>
        ⟸
        Value: <Empty>
     ⟸
     Value: <Empty>
   ⟸
   Value: <true>
⟸
Value: <true>

<true>
```

Figure A.9. The trace generated in solving the `empty-level?(bills-car)` problem task.

A.6 Extending built-in methods

Noos provides a collection of basic built-in methods. Nevertheless, some applications may require specific elementary methods that are not provided. Noos allows a simple way to incorporate new built-in methods using the `Define-Built-In` macro and Lisp expressions. This macro allows to define a new built-in method by means of specifying a name for the method, the set of required features of the method, and a Lisp expression performing a specific combination of required features. The syntax is the following:

`(define-built-in` *name* `(`*param1* … *paramn*`)` *lisp-expression*`)`

where *name* is the name of the new built-in method we are defining; *param1* … *paramn* are the names of the required features for the built-in; and *lisp-expression* is the expression that implements the built-in method. Inside the lisp expression values of required features can be accessed by using the names of required features as variables.

For instance, the `subtract` built-in method is defined as follows:

```
(Define-Built-In Subtract (amount minus) (- amount minus))
```

Moreover, we have to specify the new description of the method in Noos defining the `subtract` term as follows:

```
(define (method Subtract)
  (amount )
  (minus ))
```

The definition of a new built-in method can be specified in a file and then loaded into the Noos environment.