The ADL Scripting Model and Component Set

Oscar Nierstrasz

Centre Universitaire d'Informatique
© Université de Genève

1.   Introduction

The Activity Definition Language (ADL) [1][4] is a language for defining coordination procedures, or *workflows*. It is a textual as opposed to a graphical language. ADL activity and procedure definitions are "compiled" into CooL code, and the latter is then compiled and linked in with the run-time system for coordination procedures, called the COP kernel.

Vista [2][3][5] is a tool for visually *scripting* together pluggable software components to construct new applications. Vista maintains the graphical presentation of components and keeps track of permissible connections between them. The interfaces and the definition of "plug compatibility" for a component set are defined in a *scripting model*.

Since workflows have a natural graphical representation, it is appropriate to investigate whether Vista can be used to provide a graphical user interface for activity definition, serving as a front-end for ADL. We present here a general-purpose scripting model and component set for ADL, in which scripted components are capable of generating the corresponding ADL code for a given workflow.

The ADL scripting model presented here is intended as a detailed example of the use of Vista. The reader should keep in mind that other scripting models, components and visual presentations are possible. Furthermore, as the implementation of the ADL component set has evolved, the scripting model and components presented here do not always correspond exactly to the current implementation.

As no user manual exists yet for ADL (only the syntax and compiler description), we shall start with a brief overview of the concepts and the language. We follow this with an example of an ADL procedure as a script and continue with the detailed description of the ADL scripting model.

2.   ADL — An Activity Definition Language

ADL is a language for defining *coordination procedures*. A (coordination) procedure specifies the flow of an number of (office) objects, such as official documents, letters, forms etc., through a number of coordinated *steps*. Each step is the responsibility of a person playing some *role*, and within each step some *action* can be performed on the concerned objects.

Each procedure (and each step) specifies a number of *input* objects and a number of *outputs*. Each of these objects is of some CooL object type. Additionally, there may be some number of objects purely local to a procedure or step which appear neither as an input nor an output.

The steps of a procedure are instances of *activity* types. The reason for this separation is that many steps of office procedures are typically identical, the only difference being the source and destination of the steps inputs and outputs and the role of the person responsible for the step. As such, it is assumed that procedures will be built from a library of largely pre-defined activities. (NB: the term "activity" is reserved for the type, and "step" for the instance.)

An activity definition specifies the name of the *activity type*, the inputs and outputs and their object types, a *precondition*, which is a CooL method, an *action* (also a method), a *postcheck* method, and a number of output alternatives, depending on the result of the postcheck method. Each alternative is named, and is associated with some subset of the output objects.

In the procedure definition, activities are instantiated to steps and assigned step names. The inputs of each step are bound to inputs of the procedure or to outputs of some other step. The source of each input is given in disjunctive normal form, i.e., it depends on *all* of some set of steps terminating, *or* some other set terminating, etc. In this way, both control flow and dataflow can be specified. Additionally within a procedure, roles are assigned to steps and a *duration* may be specified. The procedure itself may have several output alternatives, depending on the outcome of its constituent steps.

Figure 1 shows an extended BNF for ADL which has been derived from the yacc source for the ADL compiler.

Keywords are in **bold**, non-terminals in *italics* and identifiers in ⟨*angle brackets*⟩. Other lexemes are within 'quotes'. Note that the keyword **ACTIVITY** within a procedure defines a step instance, whereas **DEF_ACTIVITY** defines an activity type.

The scripting model that we present is for a simplified ADL without roles, durations, preconditions or postchecks. These are straightforward to add, but contribute nothing to the exposition of the scripting model.

3.   The Common Example

Before presenting the ADL scripting model, let us look at an example of a script of an ADL procedure. In Figure 2 we see the script of the common example given in [1]. It consists of seven steps (called A, B, C, etc.) connected by workflow links. The circles (which we call "places") encapsulate the flow of office objects between steps. They correspond to the *InputLink* non-terminal in the ADL grammar. All of the steps except step B are instances of the same ADL activity, adl_print1. Step B is an instance of adl_print2. The step name, activity type and action are shown for each step. The inputs and outputs of a step are shown as small squares, respectively on the top or bottom of the perimeter of the step. The output alternatives are shown as subdivisions of the lower part of the step, and are labeled if there is more than one. Step B, for example, has output alternatives "success" and "failure."

Root ::= *SingleDefinition**

SingleDefinition ::=
    DEF_PROCEDURE ⟨*proc_name*⟩
        [**ROLE** { ⟨*role_name*⟩ { **OF** ⟨*role_name*⟩ }* [**OF INITIATOR**]
            | **INITIATOR** } [';']]
        [**DURATION** ⟨*integerconst*⟩ [';']]
        [**OBJECTS** { *ObjVarList* ':' ⟨*object_type*⟩ }+
            [**INPUT** *ObjVarList*]
            [**OUTPUT** { *InputLink*+ | { **ALTERNATIVE** ⟨*alt_name*⟩ *InputLink*+ }+ }]]
        { **ACTIVITY** ⟨*activity_name*⟩ ':' ⟨*activity_type*⟩
            [**ROLE** { ⟨*role_name*⟩ { **OF** ⟨*role_name*⟩ }* [**OF INITIATOR**] | **INITIATOR** } [';']]
            [**DURATION** ⟨*integerconst*⟩ [';']]
            [**INPUT** *InputLink*+]
            [**OUTPUT** { *ObjVarList*
                | { **ALTERNATIVE** ⟨*alt_name*⟩ [*ObjVarList*] }+ [*ObjVarList*] }]
        END_ACTIVITY }+
    END_DEF_PROC [⟨*filesep*⟩]
|
    DEF_ACTIVITY ⟨*activity_type*⟩
        [**OBJECTS** { *ObjVarList* ':' ⟨*object_type*⟩ }+
            [**INPUT** *ObjVarList*]
            [**OUTPUT** { *ObjVarList* | { **ALTERNATIVE** ⟨*alt_name*⟩ *ObjVarList* }+ }]]
        [**PRECONDITION** *MethodCall*]
        [**POSTCHECK** { *MethodCall* | { ⟨*alt_name*⟩ **WHEN** *MethodCall* }+ [*MethodCall*] }]
        ACTION ⟨*action_method_name*⟩
    END_DEF_ACT [⟨*filesep*⟩]

ObjVarList ::= ⟨*obj_var*⟩ { ',' ⟨*obj_var*⟩ }*

InputLink ::= [*ObjVarList*] **FROM** *OrBranch* { **OR** *OrBranch* }*

OrBranch ::= *OrBranchElement* { ',' *OrBranchElement* }*

OrBranchElement ::= [⟨*alt_name*⟩ **OF**] ⟨*activity_name*⟩ | **OUTER_WORLD**

MethodCall ::= ⟨*method_name*⟩ '(' [*ObjVarList*] ')'

Figure 1 ADL Grammar

The procedure consists of the collection of linked steps, input (outer) and output regions, local object variables (x, connected to outer, and y, connected inside each place), and a name ("common example"). The places connecting outputs to inputs serve as "switches" — the upper part of a place may be subdivided into alternatives, each of which waits for all of the source steps to be complete before releasing an office object. It then triggers all connected steps waiting for input. For example, steps C and D are both triggered upon the success of step B. Step F is triggered after *both* C and D are done. Step G is triggered when *either* F or E terminates.

The diagram of Figure 2 is a simplified script, not showing all connections between components. The components of the ADL scripting model are code generation components, that is, they know how to generate their ADL specifications. The procedure component knows how to generate the complete specification of a procedure as a function of its subcomponents.
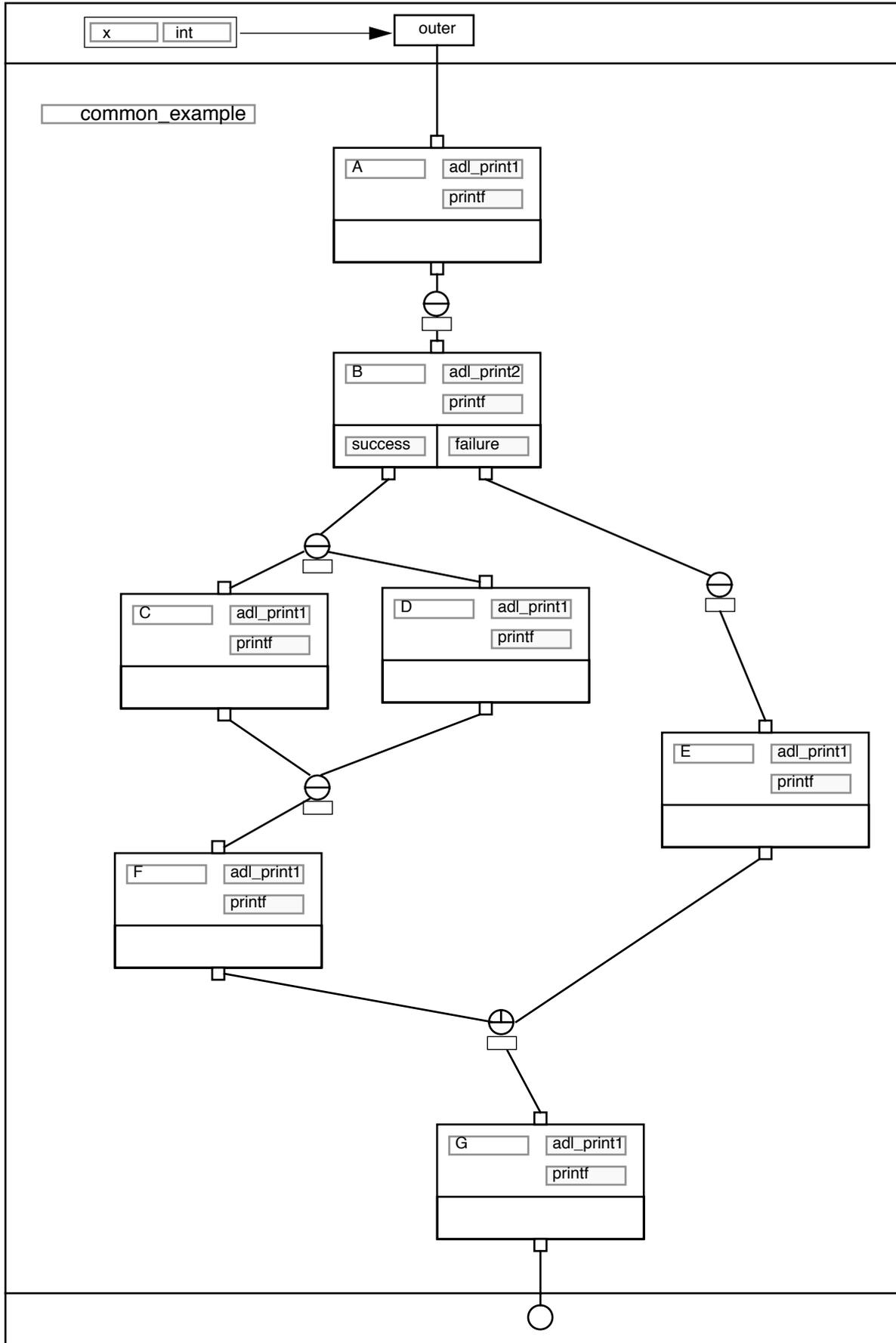
Figure 2 The Common Example as a script

4.   The ADL Scripting Model and Component Set

Vista components have *output ports*, where values are made available, and *input ports*, where values are consumed. When a connection is made between two such ports, the value available at the output port propagates to the input port. Ports are *typed*, and connections may only be made between *compatible* ports. A *scripting model* is the specification of the possible input and output port types and the rules defining port compatibility.

Vista scripting models are essentially language independent and can support a variety of computational models. Dataflows are naturally supported since each component can compute its outputs as a function of its inputs. On the other hand, object-oriented scripting is also well-supported, in which case dataflow occurs at connection time, but not necessarily at run-time.

In an object-oriented scripting model, port types represent object types, and the values propagated are typically object identifiers (oids). Each component will generally have a single output port where it makes its own oid available. Input ports represent the need for services to be provided by external objects. When a connection is made, an oid is propagated and the client stores the supplier's oid in an instance variable. The services provided can then be accessed at a later time, whenever the client needs them to fulfill a request.

The ADL scripting model is an object-oriented one in which each component represents part of an ADL procedure, and provides as services the ability to generate one or more fragments of the ADL grammar. Since each component has just one output port, we will not show them explicitly here, but make connections directly between input ports and components (as in Figure 2).

The scripting model for ADL consist of the following port types: procedure, proc_alt, proc_alts, out_port, act_alt, s_act_alt, act_alts, link_alt, activity, place, obj_var, in_port, outer. There are additionally some port types for sequences, such as seqOf_out_port, etc.

A *Script as Component* (SAC) encapsulates a script, specifying some subset of the scripts' ports to be exported as ports of the new component. In the ADL component set, activities are SACs that are instantiated as steps.

The components listed below provide, as services, the capability to produce some code fragment of an ADL definition. Each component is a template, including some standard text (such as keywords) and some variable text to be provided by some other components.

In the following, we give the (output port) *type* of each component, the services provided, the input ports, and the type of the component the inputs may be connected to. The arrows indicate the use of service, i.e., they go from an input port to the component that will provide the service. (Recall that during scripting the flow of information to establish the connection is in the opposite direction, since the oid of the service provider must be stored in an instance variable of the client.)

We will indicate where the descriptions differ from the actual implemetation.

The ADL Code Generation Component Set

An *object variable* is a component that knows its variable name and its object type name. It is capable of generating either its name (obj_var) or its declaration (obj_dec):

Component:	Object variable
Type:	obj_var
Services:	obj_dec, obj_var
In:	OName : text
	OType : text

We will represent an obj_var as follows:

ON0	OType
OName	OType

An *activity* component is a SAC that can be further bound within a procedure. So, we do not have a separate "step" component, but we just instantiate activities when we need steps. The component can generate either its activity declaration (act_def) or its step definition (step_def). To define an activity, the ActType, the ObjVars, the InPorts, the OutPorts and the Action must be given. To define a step, we must instantiate the activity, then bind the StepName, and we must bind the InPorts and OutPorts to the "places" defined within a procedure[1]:

Component:	Activity/Step
Type:	activity
Services:	step_def, act_def
In:	StepName : text
	ActType : text
	ObjVars : seqOf_obj_var
	InPorts : seqOf_in_port
	OutAlts : act_alts
	Action : text

An activity as a script is shown in Figure 3. (We do not yet show all the connections for internal components.)

Each in port of an activity must be bound to some local object variable, and, when the activity is instantiated, it must be linked to some place of the procedure. It is capable of reporting its local variable binding (local_var) and its connection to other steps (from_link):

Component:	In port
Type:	in_port
Services:	from_link, local_var
In:	ObjVar : obj_var
	Place : place

An activity alternative has an optional name and some sequence of out ports. It is capable of generating its output definition (output_def) as part of an activity definition, and its flow to other steps (proc_var_links) as part of a step within a procedure definition. There are two ver-

---

1. *Places* encapsulate the flow from out ports to in ports within a procedure. They are not defined as part of ADL, but are an essential feature of the scripting model.
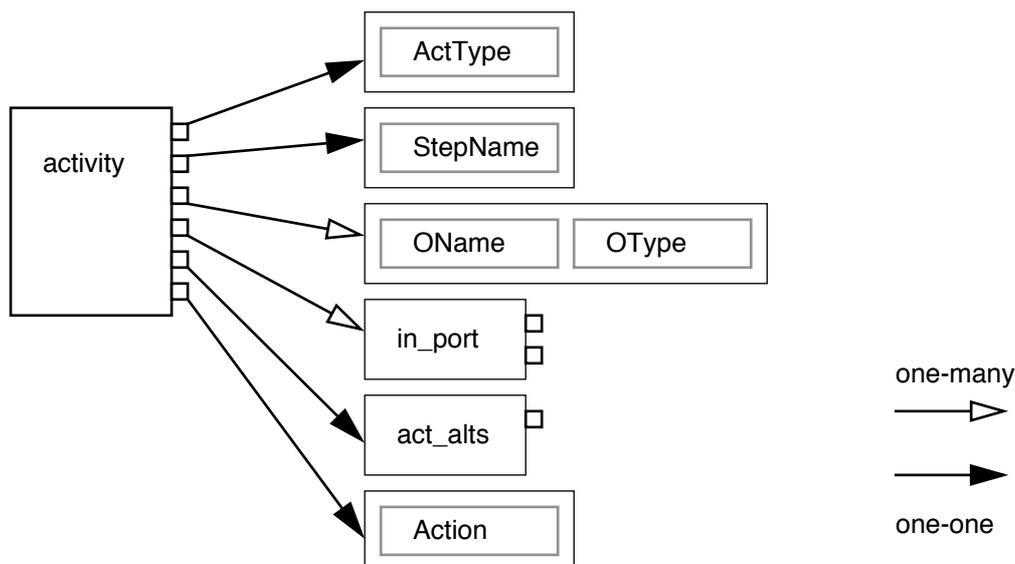
Figure 3 An activity as a script

sions of this component, depending on whether only a single alternative or multiple ones are desired. The single alternative version is a subtype of act_alts, since it is also capable of stating which alternative of which step it is (alt_spec). In the multiple alternative case, this is handled by a sub-component, act_alt.

Component:	Single activity alternative
Type:	s_act_alt < act_alts, alt_spec
Services:	proc_var_links, output_def, alt_spec
In:	StepName : activity
	OutPorts : seqOf_out_port

Component:	Multiple activity alternatives
Type:	act_alts
Services:	proc_var_links, output_def
In:	Alts : seqOf_act_alt

Component:	Activity alternative
Type:	act_alt < alt_spec
Services:	alt_spec, alt_var_links, output_vars
In:	StepName : activity
	AltName : text
	OutPorts : seqOf_out_port

The components are shown in Figure 4. Note that the alternatives are all connected to the *same* StepName component that is defined within the step.

An out port is able to generate its output specification (out_spec) within the activity definition, its connection to other steps (proc_var), and its variable type (local_var):
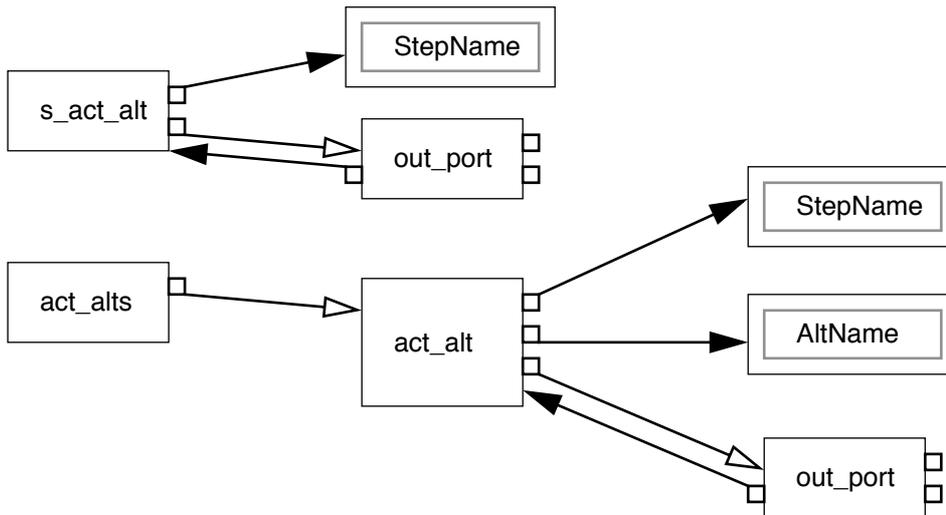
Figure 4 Activity alternatives

Component: Out port
Type: out_port < outer
Services: out_spec, proc_var, local_var
In: Alt : alt_spec
                      ObjVar : obj_var
                      Place : place

We can now re-draw activities and steps as SACs (Figure 5). The internal bindings of inputs
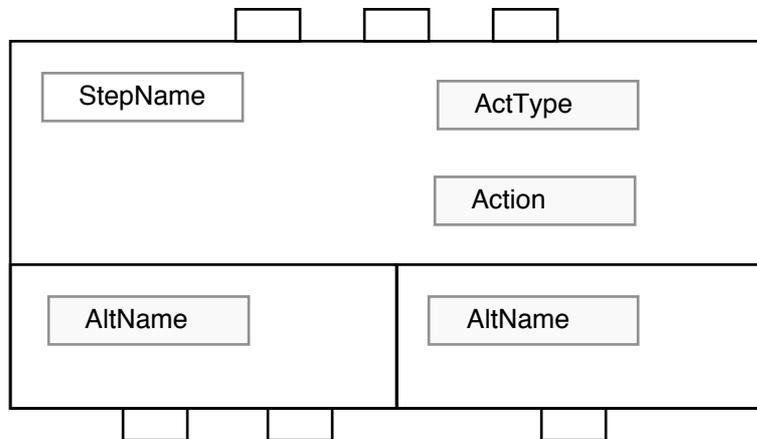

Figure 5 Activities as SACs

and outputs to local object variables are hidden inside the SAC. The activity type and the alternative names are shown, but are already bound, so are not modifiable. In practice, we will only have activities with either a single output alternative or two alternatives. In the implementation, the fact that a SAC is defined inside a window means that certain links between the components are not necessary.

The in and out ports of a step must each be bound to a place of the procedure. The place records which object variable of the procedure is used for objects that flow between steps, and it records the sources (out ports or the outer world) that objects may come from:

Component:	Place
Type:	place
Services:	output_link, var_link
In:	ObjVar : obj_var
	LinkAlts : seqOf_link_alt
Component:	Link alternative
Type:	link_alt
Services:	from_def
In:	OutPorts : seqOf_out_port

Note that a place is not directly connected to any in port, but it is the in ports of steps that are connected to places. This is purely an artifact of the ADL syntax and has nothing to do with
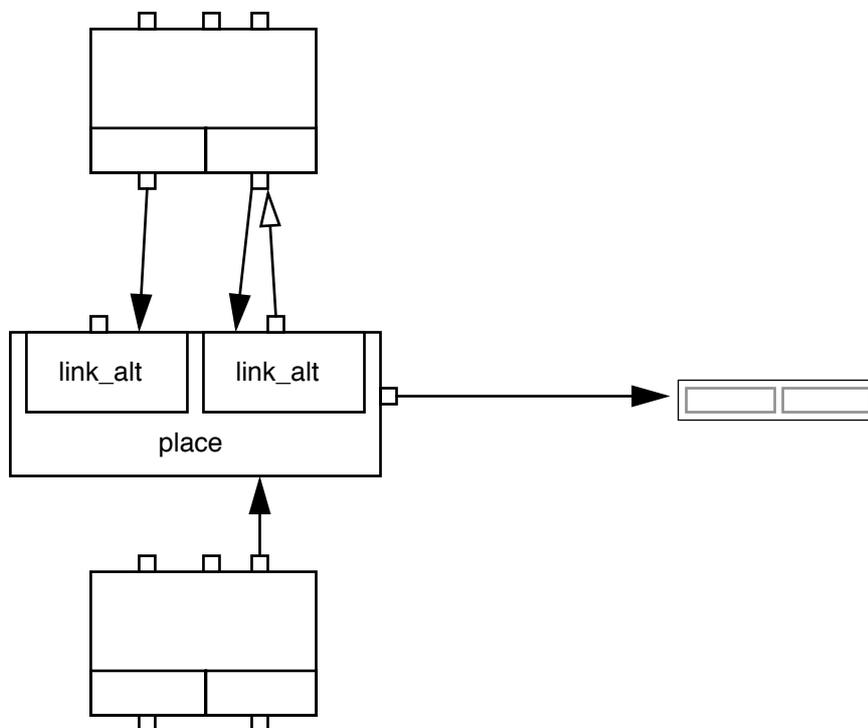

Figure 6 Places connected to steps

the semantics of coordination procedures. (In the grammar, it is with the inputs to a step that one specifies where the input comes from, not with the outputs where they go to. The outputs only need to know what object variable of the procedure the output goes to.) In Figure 6 we can see how places connect to steps. Each in and out port is connected to exactly one place, but each place may gather objects from a number of alternative sources. In practice we shall have only two kinds of places: those with one and those with two link alternatives. We will use the visual representation shown in Figure 7. In the implementation, the object variable associated with a place is internal to the place.
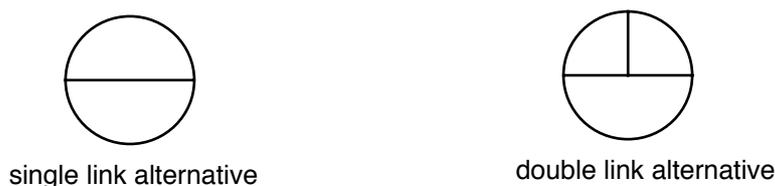
single link alternative   double link alternative

Figure 7 Visual presentation of places

The service provided by a procedure definition component is called proc_def. To complete a procedure, its name must be provided, the sequence of typed object variables, the procedure inputs, the procedure alternatives, the set of steps, and the set of activity types from which the steps are instantiated:

Component:	Procedure definition
Type:	procedure
Services:	proc_def
In:	ProcName : text
	ProcVars : seqOf_obj_var
	InVars : seqOf_obj_var
	ProcAlts : proc_alts
	Steps : seqOf_activity
	ActTypes : seqOf_activity

There are two possible ways of specifying procedure alternatives, depending on whether there is one or more. If there is only one alternative, it may be unnamed. Otherwise each alternative must be explicitly named:

Component:	Single procedure alternative
Type:	proc_alts
Services:	proc_alt_def
In:	Places : seqOf_place

Component:	Multiple procedure alternatives
Type:	proc_alts
Services:	proc_alt_def
In:	Alts : seqOf_proc_alt

Component:	Procedure alternative
Type:	proc_alt
Services:	proc_alt_links
In:	AltName : text
	Places : seqOf_place

In the implementation, procedure alternatives are represented by Result components that are linked to the steps that produce the results for the procedure.

Inputs to a procedure are specified by linking the alternatives of a place to an instance of outer instead of to the out port of a step:

Component:	Outer world
Type:	outer
Services:	out_spec

5. Concluding remarks

The ADL scripting model and component set we have presented is a general-purpose one that can be used to fully script any ADL procedure (that is, if extended to handle preconditions and roles). The components are purely concerned with generating the text of the corresponding ADL specification. This work can now be taken in several directions:

- Exploring with different visual presentations for steps and procedures. For example, it is perhaps desirable to eliminate the explicit appearance of places by providing a slightly different interface to steps.

- Specializing the ADL scripting model for different kinds of users. It should be possible, for example, to script ADL procedures from pre-packaged activities that have been directly coded in ADL.

- Extending the functionality of the ADL components, for example, to support simulation of procedures or to perform some rudimentary analysis.

- Extending the functionality of Vista. Bi-directional links and flexible placement of ports would improve appearance of scripts.

- Experimenting with a general-purpose scripting model for code generation based on "visual macros."

The scripting model presented here and the visual presentations of scripts, components and SACs shown are close, but not identical to that of the current implementation of the ADL scripting model. Please consult the demo and the user guide (forthcoming) for details.

Vista provides support for scripting models, and performs run-time type-checking when links are made. The types of the implementation are essentially the same as those given in the descriptions of the components. Again, please consult the user guide for details on scripting model support.

Acknowledgements

The ADL components were implemented by Vicki de Mey. The visual presentations were developed by Ino Simitsek. The scripting model support for Vista was designed and implemented by Betty Junod and Serge Renfer.

References

[1]   A. Graffigna, J. Li, J. Marti, G. De Michelis, J. Monguió, C. Simone, M. Tueni and H. Wiegmann, "ADL Syntax Description," ITHACA Report Nixdorf.90.U.2.#7, Siemens Nixdorf Informationssysteme AG, Paderborn, Dec 31, 1990.

[2]   B. Junod, V. de Mey, S. Renfer, "Vista Implementation," ITHACA.CUI.91.Vista.#1, Centre Universitaire d'Informatique, University of Geneva, Oct 1991, Draft.

[3]   B. Junod, V. de Mey, S. Renfer, "Vista User's Guide," ITHACA.CUI.91.Vista.#2, Centre Universitaire d'Informatique, University of Geneva, Oct 1991, Draft.

[4]   J. Li, "ADL and Its Compiler," ITHACA Report Bull.91.U2.#3, Bull SA, Massy, France, June 28, 1991.

[5]   O.M. Nierstrasz, D. Tsichritzis, V. de Mey and M. Stadelmann, "Objects + Scripts = Applications," Proceedings, Esprit 1991 Conference, Kluwer Academic Publishers, Dordrecht, NL, 1991, pp. 534-552.