Agents Classes for Managing Dialogue Control Specification Complexity

Anthony Savidis and Constantine Stephanidis
Institute of Computer Science,
Foundation for Research and Technology-Hellas (FORTH),
Science and Technology Park of Crete,
P.O. Box 1385, GR 711 10, Heraklion, Crete, Greece,
Tel: +30-81-391741, Fax: +30-81-391740
E-mails: as@csi.forth.gr, cs@csi.forth.gr

ABSTRACT
The organization of interface implementation software by means of agents is an elegant model for managing the interface construction complexity. Agents have their own local control, may manage arbitrary collections of interaction objects, and may communicate with, control, or affect other agents. The explicit realization of the agent model in dialogue specification languages is currently restricted. The I-GET interface specification language (for the I-GET UIMS) encompasses specification facilities directly reflecting the software agent model; these facilities and some key implementation issues will be presented and discussed. The agent-based specification kernel of the I-GET language provides a framework that could be combined with various other interface specification languages.

KEYWORDS: Dialogue specification, software agents, dialogue control.

1 INTRODUCTION

Software agents in the context of dialogue control are considered as independent threads of execution maintaining their own main loop, preserving their own local state, communicating with other agents, managing arbitrary collections of interaction objects, and controlling or being controlled by other agents. In the framework of the agents model, only hierarchical controlling relationships are allowed (i.e. cyclic control paths are not permitted). The agents model for software organization is particularly suited for *reactive* systems in which some of the main properties are: conditional execution, state maintenance, concurrent progress, continuous operation, and reaction to asynchronous incoming external messages.

Interactive systems are considered as instances of the general class of reactive systems. In this context, the agent model provides a powerful approach for modular organization of dialogue management software, by supporting key control structures like distributed / localised, hierarchical and centralized schemes. Previous work in this field mainly falls into three categories: (i) Generic agent-based models for interactive software organization. One well known such model is PAC [3], according to which interactive software can be structured in hierarchical organization of agents that are composed of three main components: Presentation, Abstraction and Control; mapping of an Abstraction to a Presentation, as well us the syntax for interactive manipulation of a Presentation, is purely the responsibility of the Control component. (ii) Explicit programming models embedded in programming languages and toolkits. The MVC (Model-View-Controller) [4] is a well known approach, which, however, is not purely based on agent organization. The three parts of an MVC structure are distinct Smalltalk classes, while the Controller does not realize all parts of control (some type of control and presentation syntax must be handled by Model and View as well). (iii) Agent-oriented dialogue specification approaches. The SERPENT UIMS [2] has provided an explicit realization of an agent-like control model in a dialogue specification language (called Slang). The agent model realized by Slang is very close to the PAC model, and it supports the specification of agent-like entities called ViewControllers (construction of hierarchies is facilitated). The activation of ViewControllers in SERPENT can be specified either through preconditions which are based on events occuring in the shared space, between the dialogue control and the application component, or through simple boolean flags. The I-GET UIMS [7] provides a dialogue specification language which realizes a pure model for agent-based

dialogue control specification (the initial version of the agent-based language kernel has been realized within the HOMER UIMS [6]). Agent hierarchies are supported, realizing hierarchical controlling schemes, while the instantiation / destruction of agents can be specified via instantiation / destruction preconditions based on: (i) conventional programming expressions (like those of the C language), and (ii) events on the application interface layer; in I-GET, shared space and typed message channels realize this layer. Additionally, the specification of parameterized agents is supported, in order to enable the construction of reusable sub-dialogues. The agent-based dialogue control specification method of the I-GET language will be presented and discussed, while some specific implementation issues in this context will be addressed.

2 STARTING WITH A SIMPLE EXAMPLE

All examples will be simple interactions, however, reflecting important control issues. The screen snapshots for these examples are provided within the Appendix, at the end. The first specification example which will discussed concerns the specification of a simple confirm box dialogue:

```
1:  bool confirm=false, opened=false;
2:  agent ConfirmBox
3:  create if (confirm==true && !opened)
4:  bool close=false;
5:  destroy if (close==true) [
6:      Window win;
7:      Button       ok(win),       no(win),
        cancel(win);
8:      Label msg(win);
9:      method ok.Pressed [
10:         // stmts for "ok"
11:         close=true;
12:     ]
13:     // methods for other two buttons
14:     constructor [
15:             opened=true;
16:             // set object attributes
17:     ]
18:     destructor [opened=false;]
19: ]
```

In this example an agent class specification is provided between lines 2 and 19. The agent class name is `ConfirmBox`. This agent class has an instantiation precondition `confirm==true && !opened` (line 3) accessing the *global* variables `confirm` and `opened`. Also, it has the destruction precondition `close==true` accessing the *local* variable `close`. Each time `confirm` or `opened` variables change, the precondition is re-evaluated, and if it is found to be `true`, an instance of the `ConfirmBox` agent class will be directly created. The role of the boolean variable `opened` is to ensure that only one instance of the confirm box is allowed at a time; within the constructor of the `ConfirmBox` agent class, the `opened` variable gains directly the value `true` (line 15), so that subsequent instantiation are disabled since the instantiation precondition would always fail.

The `opened` variable is reset again to `false` within the destructor so that a new instantiation can be allowed (line 18). If it is required to enable multiple instantiations of the `ConfirmBox` agent, it suffices to put `confirm==true` (line 3) as an instantiation precondition, remove line 15 and line 18 (only the assignment), and simply set `confirm` to `true` each time an agent instance would be required. It should be noted that instantiation preconditions can only engage non-local variables. Regarding destruction, each time the local variable `close` of any particular instance gains the `true` value, that agent instance will be destroyed. In line 5, the opening square bracket `[` matches the closing square bracket `]` of line 19, and indicate the main *body* of the `ConfirmBox` agent class, enclosing items *local* to the class.

Between lines 6 and 8, five interaction objects are declared; some of the details for declaration of objects from the original syntax of the I-GET language have been omitted for clarity. Each instantiation of an agent class causes instantiation of all interaction objects declared within the body of that agent class, while the destruction of an agent instance will also destroy the associated object instances. For the activation of the confirm box (i.e. create a `ConfirmBox` agent instance) it is only required to cause `confirm` variable to take the true value (either via a direct assignment, or indirectly, e.g. through pointers). Between lines 9 and 12, the callback code for the `Pressed` method of the `ok Button` is provided (again the original I-GET language syntax has been simplified for clarity). Within this code, the specific actions for the *Ok* user selection are specified (programming statements) while the final statement closes down the confirm box (line 11). Similar callback implementations may follow for the other two `Button` objects.

The destruction precondition is optional for agent classes. In case that such a precondition is not provided, it is possible to specify destruction directly through a statement. For instance, the statement `destroy myagent;` will directly destroy the instance in which this statement is called (`myagent` is similar to the *this* pointer in C++). It is also possible to extract the agent instance pointer from interaction objects; for example, `destroy ok.myagent;` in the previous example will have

the same effect. Both `myagent` and `<obj>.myagent` are expressions of a specific built-in type `daid` (dialogue agent identifier); `daid` variables can be passed as arguments to the `destroy` operator. As it will be shown later on with more complex examples, this scheme provides an appropriate way for the specification of external management (i.e. outside the body of agent classes) of the destruction of agent instances, which is particularly appropriate for building agents that control other agents (i.e. higher level dialogue control).

3 MAKING REUSABLE PARAMETERIZED AGENTS

There are different instantiation policies for agent classes: (i) *precondition-based* instantiation, where agents are automatically instantiated on the basis of instantiation preconditions (as it has been shown in the first example), and (ii) *call-based* instantiation, where instead of instantiation preconditions, agent classes are defined with an arbitrary set of formal arguments, while instantiation is explicitly specified via instantiation statements when required (by passing real values to the formal arguments). The call-based approach is similar the instantiation mechanism in C++ where explicit declaration statements are supported, while real arguments can be passed upon instantiation via the class constructor. Even though precondition-based instantiation is more "declarative" in nature, it restricts the reusability of the associated agent classes. The primary reason is that instantiation preconditions of agent classes necessarily access non-local variables, thus limiting the ability of using agent classes as generic dialogue patterns, applicable in different specification contexts. Additionally, it is necessary to support parameterization of agent classes in order to support the specification of reusable agent *black boxes*. The following example shows a reusable call-based confirm box agent class:

```
1:  agent ConfirmBox (
2:  bool* okDone,
3:  bool* noDone,
4:  bool* cancelDone,
5:  string msgText
6:  ) [
7:      // object declarations here
8:      method ok.Pressed [
9:          *okDone=true;
10:         destroy myagent;
11:     ]
12:     // methods for other buttons here
13:     constructor [
14:         msg.text=msgText;
15:         // set other attributes
16:     ]
17:     destructor []
18: ]
```

Between lines 2 and 5, the parameters passed to the `ConfirmBox` agent are shown. The three pointer variables are used for returning values to the calling context. These pointers correspond to boolean flags which are set within the callbacks for the corresponding buttons (for instance, `*okPressed` is set within the method implementation of `ok button` between lines 8 and 11). The `msgText` carries the context of the `msg Label` assigned in line 14. The way in which the confirm box agent class can be used in any specification context follows:

```
1:  bool okDone=false;
2:  bool noDone=false,
3:  bool cancelDone=false;

4:  okDone: [
5:      // actions for Ok
6:  ]
7:  // monitors for the other two flags here

8:  create ConfirmBox(
        &okDone,
        &noDone,
        &cancelDone,
        "Replace target file?"
    );
9:  // next statement here
```

The declaration of the boolean variables is shown between lines 1 and 3. The addresses of these variables are passed within the instantiation statement shown in line 8. An explicit agent instantiation via a statement (as in line 8) can be considered as creating an additional thread of execution (i.e. the agent) realizing its own control and main loop. Consequently, the statement of line 8 is not similar to a function call, but rather to a *create process* system call. As a result, it is meaningless to access within the statement in line 9 (or any to follow) the boolean variables for gaining the user selection (since selection at that point may not have been performed yet).

Asynchronous notification for modification of the supplied boolean flags can be received and handled via *monitors* in I-GET. A monitor is a code segment (i.e. a block) attached to an Lvalue, that is automatically called at run-time when the associated Lvalue changes. Between lines 4 and 7, the specification of the monitor for the `okDone` flag is provided. The code for this monitor is executed when the content of the `okDone` flag changes, which is only possible in the context of the confirm box agent, within the method for the `ok Button`, through the `*okDone=true;` statement in line 9 of the agent `ConfirmBox` specification (such indirect changes, involving arbitrary

pointers, are also detected by I-GET). Similar monitors need to be specified for the other two flags (line 7).

4 CENTRALIZED CONTROL SCHEMES

The agent model primarily supports distributed control, since agents "work" as independent processes, each performing its own local control. To demonstrate the realization of centralized control schemes via agents, the following interface design artifact is considered: a number of various interactive services is provided to the user; the user is enabled to start / end an interactive service; multiple instances of the same interactive service are not allowed; service management is provided to the user via a single control panel. In this artifact, the requirement is to support a distributed localized control of the various service dialogues, while maintaining a centralized management of the initiation / cancellation interaction for all services. The following specification meets this requirement:

```
1:  bool servStatus[3]=
2:  [false,false,false];

3:  agent Service0
4:  create if (servStatus[0]==true)
5:  destroy if (servStatus[0]==false) [
        // agent body for service A dialogue
        // here
    ]
// similar agents for rest services here
//
6:  agent ControlPanel
7:  create if (true) [
8:      Window panel;
9:      Toggle serv0(panel);
10:     Toggle serv1(panel);
11:     Toggle serv2(panel);

12:     method serv0.StateChanged [
13:             servStatus[0]=serv0.state;
14:     ]
        // methods for rest toggles here
        //
15:     constructor [
16:         serv0.state=serv1.state=
            serv2.state=false;
        ]
        destructor []
    ]
```

The dialogue for each service is realized via an associated agent class. In the previous example, there are three services and three corresponding agent classes. An instance of such an agent class (e.g. `Service0`) would provide the interactive realization of the corresponding service. The instantiation / destruction of the service-specific agent classes is controlled centrally via the `servStatus[]` array of activation flags (line 1), having initially all flags `false`. All service-specific agent classes have similar instantiation and destruction preconditions (i.e. `servStatus[i]==true`, `servStatus[i]==false`, where `i` is the service number), shown in line 4 and line 5. The main control panel dialogue is provided by a single independent agent class named `ControlPanel`. One instance of this agent class is created upon start-up since the precondition is `true`, and also it is not possible to have other instances created (i.e. since its precondition is constant, it will never be re-evaluated). This agent instance provides interaction through which the user may have control on when a service becomes interactively available or when it is closed.

There are three `Toggle` buttons provided by the `ControlPanel` agent instance (lines 9-11), each corresponding to a particular service. When the boolean attribute `state` changes (i.e. the user "presses" the object, and the callbacks for the method `StateChanged` are called, lines 12-14), the status of the corresponding service (e.g. `servStatus[0]`, for `Service0` agent class) is set to the value of the `state` attribute (line 13, for `Service0` agent class). If this value is `true`, an instance of the corresponding agent class is created since the instantiation precondition succeeds (`servStatus[0]==true`, for `Service0` agent class), else the existing instance is destroyed (changing the `Toggle` state to `false` means that it has previously been `true`, thus an instance had been already created). Enforcing that only one agent instance per service-specific agent class is active, is handled in a different manner with respect to the confirm box example, where a dedicated flag (i.e. `opened`) has been employed. In this example, the use of `Toggle` buttons enables the user to only activate / close a service (i.e. there is no path towards two successive activations) directly by changing the state of the corresponding `Toggle` buttons.

5 CONSTRUCTING AGENT HIERARCHIES

Until now, the only constructs that have been defined within the body of agent classes in the previous examples have been: method implementations (i.e. callback functions), interaction objects, and the constructor / destructor standard functions. Additionally, it is allowed to specify agent classes within agents. This feature results in the construction of agent hierarchies. The specification of an agent class *X* within another agent class *Y* defines a *controlling* relationship, rather than a *part-of* relationship: instances of the agent class *X* are created within the context of instances of the *Y* class, however, they do not constitute components of the instances of agent class *Y*; also, if there is no

instance of *Y* class, then no instance of *X* class can be created.

We will use an example scenario to show both the main specification aspects, as well as the practical value for agent hierarchies in dialogue specification: an interactive document processor is to be developed; the system provides a main window which encompasses the text editing area and some objects to activate operations for font-style setting, spell checking, etc. The following specification serves our needs (various specification details have been removed):

```
1:  agent DocProcessor
2:  create if (true) [
3:      Window  win;
4:      Toggle      font(win);
5:      Toggle      spell(win);
        TextEditedit(win);
6:      agent   FontStyleSet
7:      create if (font.state==true)
8:      destroy if (!font.state) [
        ...
        ]
9:      agent   SpellCheck
10:     create if (spell.state==true)
11:     destroy if (!spell.state) [ ...
        ]
    ]
```

The document processor is realized via an agent class having a constant `true` instantiation precondition (line 2); this constant precondition will result in a direct instantiation of the `DocProcessor` agent class. The dialogue for all operations supported by the document processor (i.e. font-style setting and spell checking) is specified via specific agent classes (i.e. `FontStyle` starting in line 6, and `SpellCheck` starting in line 9) defined in the context of the `DocProcessor` agent class. `Toggle` buttons are declared, one per supplied operation (i.e. `Toggle font` in line 4, and `Toggle spell` in line 5); the `state` attribute for each such button is used within the instantiation precondition of the agent classes providing the dialogue for the associated operation (lines 6 and 9). Hence, the font-style operation which is provided by the `FontStyle` agent class, has the `font.state==true` instantiation precondition. The destruction preconditions are similar (lines 8 and 11).

Agent hierarchies provide a clean way to specify sub-dialogues which correspond to distinct user task sub-hierarchies (the latter resulting from user task identification phase, part of the dialogue design process). A hierarchy of agent classes needs to be constructed, which is isomorphic to the user task hierarchy, while task operators such as *repetition* (both *, and **+**), *exclusive selection*, *sequencing* and *concurrency* [5], [1] can be easily reached via instantiation and destruction preconditions (the fundamental set of arithmetic and boolean expressions is supported for preconditions).

An embedded agent is called a *child* agent, while the direct owner agent is called *parent* agent. All hierarchically higher agent classes are called *ancestors*. Parentless agent classes are called *toplevel* agents. Call-based parameterized agents can be only toplevel agents in I-GET for the following reason: considering that all child agents are not visible outside their parent agent scope, usage of child agents is restricted to their parent agent specification context (i.e. breaking the principle of reusability); hence, in this situation parameterized children agents cannot support reusability, which has been the primary purpose of introducing parameterized agent classes. Nevertheless, it is legal to have the toplevel parent of an agent class hierarchy being a call-based agent class.

Scope rules

Child agent classes inherit the scope of all their ancestor agent classes and thus have access to all constructs that can be directly referenced by any ancestor agent class. For instance, consider the example below (preconditions, constructors and destructors have been removed for clarity):

```
1:  agent X [
2:      int i;
3:      agent Y [
4:          int j;
5:          agent Z [ ]
        ]
    ]
```

Within agent `Y`, the local variable `i` of agent `X` (line 2) can be accessed. Similarly, variable `j` (line 4) can be accessed from agent `Z`, and also variable `i`, since `i` is accessible from the parent of agent `Z` (i.e. `Y`). At run-time, multiple instances of a child agent class may exist. Each such instance is associated with the specific parent agent instance *through which* it has been instantiated (to be explained in Section 8), while all run-time references to non-local items (belonging to instances of ancestor agent classes) "pass" through this specific parent instance. Priority is given to local items in case of name collisions. Accessing non-local items in such cases is made possible via scope resolution (similar to C++, with the difference that in C++ it applies to class inheritance hierarchies, while in I-GET it applies to agent controlling hierarchies). The following example shows how scope resolution applies to different types of items within agent classes (some details are removed for clarity):

```
1:  agent X [
2:      Button a; int i;
3:      void f() [ i++; ]
4:      agent Y [
5:          Button a; int i;
6:          void f() [ i--; ]
7:          method X::a.Pressed []
8:          method a.Pressed []
9:          constructor [
10              i=X::i;
11:             X::f();
12:             f();
        ]
    ]
```

Scope resolution has the syntax *<agent class>::<item name>*. The name of the agent class should be always a hierarchically higher agent (agents should have unique names within any hierarchy path) with respect to the particular agent class in which scope resolution operator is used. As a result, `X::i` (line 10), `X::f` (line 11), and `X::a` (line 7) when used within agent class `Y`, refer to the `i` (line 2), `f` (line 3), and `a` (line 2) of agent class `X`.

6 PRECONDITIONS BASED ON APPLICATION INTERFACE EVENTS

The I-GET application interface method supports both the shared space model, where arbitrary data structures can be exported (in shared space), and the model of typed communication channels, which facilitates the establishment of an arbitrary number of strongly typed links for information exchange. The application interface specification in I-GET involves the specification of shared structures as well as the definition of typed channels. A simple example follows:

```
1:  enum Commands=[rlogin,ftp,rsh,mail];
2:  channel CommandsChannel of Commands;
3:  struct Employee [
        string name, role; real salary;
];
4:  shared Employee;
```

One channel, named `CommandsChannel` (line 2) carrying `Command` type data (line 1), and one shared type (line 4), the `Employee` structure (line 4), have been defined. The I-GET language provides statements in dialogue specification: (i) to create, modify, read and destroy shared data structures, and (ii) to send typed messages to specified channels. Software libraries of similar purpose are provided for the application developers.

Additionally, the I-GET language supports preconditions which correspond to notifications for the following application interfacing events: (a) creation, destruction and modification of shared structures in shared space, and (b) incoming messages (to the dialogue control component) through typed channels. The following example shows the specification of an agent which has instantiation and destruction preconditions which correspond to such types of events:

```
1:  agent DisplayEmployee
2:  create if (shcreate (Employee emp,
    empid))
3:  destroy if (shdestroy (empid)) [
4:      Window win;
5:      Label name(win);
6:      Label role(win);
7:      Label salary(win);
8:      constructor [
9:          name.text="Name:"+emp.name;
10:         role.text="Role:"+emp.role;
11:
                salary.text="Salary:"+emp.sa
            lary;
        ]
    ]
```

In line 2, the instantiation precondition defines that an agent instance will be created each time an event for the shared creation (define via the `shcreate` keyword) of an `Employee` structure is received from shared space. The instantiation precondition has the following syntax: the shared type is provided (e.g. `Employee`) followed by the name of a variable (e.g. `emp`) to store the created `Employee` object, when the event is detected at run-time; I-GET will update automatically the `emp` variable, each time the newly created shared object is modified; finally, the name of a variable (e.g. `empid`) to hold a returned (by I-GET) internal identifier, uniquely associated with the created shared object, is provided; this identifier can be used for further manipulation of the shared item within the agent specification. Both of these implicitly declared variables, i.e. `emp` and `empid`, are made accessible within the agent as normal local variables. The destruction precondition concerns notification for destruction of the specific shared instance that has been created; this is accomplished by passing the `empid` variable within the `shdestroy` event (line 3). In the body of the agent, some `Label` interaction objects (lines 5-7) for displaying the fields of the `Employee` record (lines 9-11) are declared.

Similar types of facilities regarding detection of events in shared space through preconditions of agent-like constructs called ViewControllers are also supported by SERPENT [2]. The following example shows the way preconditions based on events on typed channels can be specified:

```
1:  agent RemoteUserCmmd
2:  create    if    (message(CommandsChannel
    cmmd)) [
    // main body
]
```

In line 2, the precondition syntax for receiving a message channel event requires the name of the channel (in this case `CommandsChannel`), and the name of a variable to place the received message (in this case `cmmd`).

Combining API event-based preconditions and expression-based preconditions

In the I-GET language, it is not allowed to combine API events and conventional expressions with boolean operators in order to formulate more complex preconditions for agent classes. However, any such complex case can be easily accomplished via the present specification facilities of the I-GET language. Consider the general case of an API event *E*, an expression *P* and a boolean operator *op*. Then primitive combined preconditions are of the form *(E op P)*. Assume that such a precondition is used as an instantiation precondition for an agent class *X*. Then, such an agent class can be defined through the help of an additional agent class specification that is dedicated on receiving the API event. The following example shows how this is made possible (*E* has been selected to be a shared creation of an `Employee` record):

```
1:  bool E_received;
2:  Employee emp;
3:  agent X_help
4:  create    if    (shcreate(Employee    emp,
    empid))
5:  destroy if (shdestroy(empid)) [
6:      ::emp := emp;
7:      constructor [
8:          E_received=true;
        ]
    ]

9:  agent X
10: create if (E_received op P) [
        // use ::emp global in this block
    ]
```

The agent `X_help` (line 3), receives the event for shared creation of an item and it forces a global variable (i.e. `emp`, declared in line 2, accessed as `::emp` for global scope resolution) to have always the same content with the implicit variable `emp` of the instantiation precondition, via the constraint in line 6. Also, it sets in the constructor (line 8) the boolean `E_received` (declared in line 1) to `true`, indicating that the event has been received. In line 9, the agent `X` has an instantiation precondition which is a simple expression engaging the `E_received` boolean and the *P* expression via the *op* boolean operator. When this precondition succeeds, the event has been received and the expression-based precondition is `true`. Within the body of the `X` agent, the global variable `emp` is used to access the API event content. In the previous example, the combined precondition involved only one API event. In case of more complex combined preconditions engaging multiple API events, a generalization of this technique can be easily applied: an appropriate global list for storing received event contents (i.e. the purpose of the `emp` variable) is necessary, while multiple auxiliary agents, each handling detection of a distinct API event, need to be specified.

7 OTHER TYPES OF CONSTRUCTS ALLOWED WITHIN AGENTS

In the context of an agent class, variables and interaction objects can be declared, while the following types of constructs can be defined: data structures, functions, method implementation callbacks, unidirectional constraints, monitors, event handlers and agent classes. Instances of all constructs owned by an agent class, apart from the children agents (since these are not *contained* by means of a "part-of" relationship), are directly instantiated when an instance of that class is created at run-time. Due to supporting agent class hierarchies, it is possible at run-time to have multiple instances of the same child agent class *accessing* (i.e. referencing) the local variables and objects of the same parent agent class instance. Assume that, within the child agent class, various constraints / monitors applied on variables / interaction objects, as well as event handlers / method callbacks for interaction objects owned by the parent instance, are defined. Then, at run-time, each child agent instance applies its own set of these constructs to the same instances of variables / interaction objects (of the single parent agent instance). In such a situation, the following rules are applied at run-time:

Multiple constraints on the same item. In I-GET, constraints are "stacked" per item at run-time, having always the most recently established constraint being the active one. An instance of a constraint is established at run-time, when an instance of the agent class encompassing that constraint is created; upon destruction of this agent instance, the established constraint instance is also cancelled.

Multiple method callback implementations for the same object. In this case, all callbacks are executed in the reverse order in which they have

been registered (at run-time). Overriding previously registered callback implementations for the same method is not directly possible in I-GET and it requires additional specifications (control flags and conditional statements in the implemented callbacks) to be achieved.

Multiple monitors defined for the same object. All registered monitors are executed, in the reverse order in which they have been registered at run-time.

Multiple event handlers defined for the same object and the same event class. Having multiple event handlers on the same object is natural. Having multiple event handlers for the same event class will cause execution of all event handlers, however, in an unpredicted order.

8 IMPLEMENTATION ISSUES

The I-GET UIMS provides a compiler for the I-GET language which translates dialogue specifications to a set of C++ implementation modules. Each agent class is translated to an appropriate C++ class. Even though in the context of interface specification a child agent class is always specified in the body of its parent agent class, upon code generation these two classes are completely separated.

Managing child agent instances

Each agent class that plays the role of a parent agent class for any particular set of *N* child agent classes, upon code generation, encompasses within the corresponding C++ agent class the instantiation preconditions of the *N* child agent classes (the way in which preconditions are translated in C++ will not be discussed in this context). Upon instantiation of an agent class, the newly created agent instance *makes active* (to be explained in next paragraph) the previously mentioned set of instantiation preconditions for child agent classes. At run-time, some child agent instances may be created as a result of these particular instantiation preconditions. In this case, these resulting child agent instances are considered to be created *through* the specific parent agent instance; the local `myparent` pointer of the child agent instances (C++ classes) is set to the parent agent instance (see Figure 1). Child agent instances always notify their parent agent instance upon destruction.

Agent code-generated structure and precondition management

A simplified view of an agent class structure upon code generation (quickly mentioned as agent C++ class) is provided in Figure 1. Each agent C++ class contains: (i) a number of members which correspond to the various items that are specified in the context of an agent class body, (ii) the destruction precondition (optional), (iii) the instantiation preconditions for its child agent classes (if applicable), (iv) a parent agent class pointer (i.e. `myparent`) which takes its value at run-time upon instantiation, and (v) a list of child agent instances, which is also maintained at run-time.

Figure 1: Run-time picture for agent instances and precondition tables.

In I-GET, two global tables of active preconditions is maintained (see Figure 1): one for expression-based preconditions and one for API event-based preconditions. The reason for having two different types of tables is related to the necessarily different way of evaluating / managing these two classes of preconditions:

Expression-based preconditions. All variables in I-GET at run-time have standard methods for notification in case of content modification (i.e. monitors). Also, all variables maintain a "precondition engagement" list, with pointers to the expression-based preconditions in the global table, in which they are involved. Each time a variable changes (e.g. directly / indirectly due to the execution of a programming statement), it notifies the precondition table to re-evaluate the necessary preconditions. This is a distributed notification algorithm and it reflects the de-centralized nature of the "events" (i.e. variable updates) that cause expression-based precondition re-evaluation.

API event-based preconditions. At run-time, events from the shared space and the message channels are centrally received within dialogue

control (i.e. API precondition control module, see Figure 1). Each API event-based precondition in the global table holds information regarding the expected event type (and also some other matching parameters). Upon receiving such events, the API precondition control identifies the appropriate preconditions from the table that need to be evaluated.

The way in which preconditions are installed / cancelled in / from the global precondition tables is defined by the following rules (these rules denote some of the standard actions that all agents perform upon instantiation and destruction):

- *Upon start-up, all instantiation preconditions of toplevel agent classes are installed in the global tables.*
- *When an instance of an agent class **X** is created, the instantiation preconditions for all the child agent classes of **X** are installed in the global tables. Its destruction precondition is installed in the global tables.*
- *When an agent instance is destroyed, the instantiation preconditions of its child agent classes are cancelled from the global tables. Its destruction precondition is cancelled from the global tables.*
- *When an agent is destroyed, all its child agent instances are destroyed.*

Translating references to non-local items within agent hierarchies

The hierarchical agent organization provides a clean scheme for accessing non-local items of ancestor agent classes in dialogue specification. However, child agent instances are not "physically" (i.e. by means of a contain relationship) included within parent agent instances; this is not the result of a development decision, but reflects the fundamentals of the supported agent model realizing hierarchical controlling relationships. Consider the following example agent hierarchy:

```
1:  agent X [
2:      int i;
3:      agent Y [
4:          constructor [
                i=23;
            ]
        ]
]
```

At run-time, any instance of the `Y` agent should "see" the `i` of the agent instance through which it has been instantiated. In this case, this `i` variable is found to be one level of hierarchy above, while in the general case, it could be referenced (via scope resolution, if name collisions apply) from any ancestor agent class. The I-GET compiler will translate (upon code generation) the `i` reference within the `Y` agent to `myparent->i`, thus correctly getting the value of the parent instance local `i` variable. In the general case, the reference is translated to (`myparent-><var>`)^n, where n is the distance between the ancestor and the agent in which reference of the variable is made (the distance between agent `X` and `Y` is defined to be 1).

9 DISCUSSION AND FUTURE WORK

The agent model is well suited to the needs of dialogue control specification for interactive systems, since it allows to easily capture and effectively express the various distributed / centralized, dynamic, concurrent and asynchronous properties of User Interface management. Also, the agent model is highly object oriented and it supports the elegant and modular organization of User Interface software. The I-GET language realizes an explicit agent model for dialogue control specification; the specification kernel presented forms a framework which can be employed in other languages as well, if appropriately adapted. One additional feature regarding the basic agent model is the ability to specify parameterized agents that can realize re-usable dialogue patterns; such special-purpose agent classes may encompass an arbitrary number of precondition-based agent classes for the realization of internal dialogue control aspects (i.e. it is more than having conventional re-usable programming classes or software modules implementing some kind of generic dialogue functionality). Also, mapping of a user-task hierarchical structure (involving task-related operators such as repetition, exclusive choice, sequence and concurrency) to a collection of hierarchically structured agent classes can be accomplished easily, and in a well defined manner, on the basis of our agent-oriented specification approach.

Some additional important features supported by the I-GET specification language, mainly reflecting OO language programming facilities, are: (i) Support for access qualifiers for all types of local (to agent classes) items. The `public` and `private` qualifiers have been borrowed from C++ with the same semantics; (ii) Introduction of pointers to agent class instances, through which all public local items of an instance can be notationally accessed. Although this last feature appears as a "programming" issue, it enhances the expressiveness for the controlling schemes since it allows external access (e.g. from a central global controller agent) to all public agent members.

ACKNOWLEDGEMENTS

This work has been partially funded by the TIDE Programme of the Commission of European Union (DG XIII), under the projects GUIB (TP 103), completed in June 1993, GUIB-II (TP 215), completed in December 1994, and ACCESS (TP 1001), to be completed in December 1996.

REFERENCES

[1]. Alexander, H. Structuring dialogues using CSP. In *Formal Methods in Human-Computer Interaction*, Harisson, M & Thimbleby, H. (Eds), Cambridge University Press, 1990, 273-295.

[2]. Bass, L., Hardy, E., Little, R., Seacord, R. Incremental development of User Interfaces. In *Engineering for Human-Computer Interaction*. G. Cockton, Ed. North-Holland, 1990, 155-173.

[3]. Coutaz, J. Architecture Models for Interactive Software : failures and trends. In *Engineering for Human-Computer Interaction*. G. Cockton, Ed. North-Holland, 1990, 137-151

[4]. A. Goldberg, D. Robson. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley Publishing, 1984.

[5]. HOARE, C. A. R. Communicating Sequential Processes. In *Commun. ACM 21, 8* (Aug. 1978), 666-677.

[6]. Savidis, A., Stephanidis, C. Developing Dual Interfaces for Integrating Blind and Sighted Users: the I-GET UIMS. In proceedings of the *CHI'95 conference in Human Factors in Computing Systems*, Denver, Colorado, May 7-11, 106-113.

[7]. Stephanids, C. Towards User Interfaces for All: some critical issues. In panel session on *User Interfaces for All: Everybody, Everywhere, and anytime*. In proceedings of the HCI International'95, Tokyo, Japan 9-14, Vol 1, 137-142.

APPENDIX - The simple specification examples.

ContainsDatafor

PostscriptOnly.

In the above screen snapshot agents are not to be confused with windows. The title of the windows has been specified to be "agent <agent>:Window win" just to indicate the class of the agent instance in which the presented windows has been declared.