Proceedings of the International Multiconference on Computer Science and Information Technology pp. 181–188
ISBN 978-83-60810-27-9
ISSN 1896-7094

A Technique based on Recursive Hierarchical State Machines for Application-level Capture of Agent Execution State

Giancarlo Fortino* and Francesco Rango
DEIS – University of Calabria, Via P. Bucci cubo 41c, 87036 Rende (CS), Italy
Email: g.fortino@unical.it, frango@si.deis.unical.it

Abstract—**The capture of the execution state of agents in agent-based and multi-agent systems is a system feature needed to enable agent checkpointing, persistency and strong mobility that are basic mechanisms supporting more complex, distributed policies and algorithms for fault tolerance, load balancing, and transparent migration. Unfortunately, the majority of the currently available platforms for agents, particularly those based on the standard Java Virtual Machine, do not provide this important feature at the system-level. Several system-level and application-level approaches have been to date proposed for agent state execution capture. Although system-level approaches are effective they modify the underlying virtual machine so endangering compatibility. Conversely, application-level approaches do not modify any system layer but they provide sophisticated agent programming models and/or agent converters that only allow a coarse-grain capture of agent state execution.**

In this paper, we propose an application-level technique that allows for a programmable-grain capture of the execution state of agents ranging from a per-instruction to a statement-driven state capture. The technique is based on the Distilled StateCharts Star (DSC*) formalism that makes it available an agent-oriented type of recursive hierarchical state machines. According to the proposed technique a single-threaded agent program can be translated into a DSC* machine by preserving its original semantics. Although the proposed technique can be applied to any agent program written through an imperative-style programming language, it is currently implemented in Java and integrated into the JADE framework, being JADE one of the most diffused agent platforms. In particular, agents, which are specified through a generic Java-like agent language, are translated into JADE agents according to the JADE DSCStarBehaviour framework. A simple yet effective example is used to illustrate the proposed technique.

I. Introduction

AGENT-BASED and Multi-Agent Systems are developed around the concept of agent, a goal-directed, computational and interacting entity which acts on behalf of another entity (or entities) [20]. Such agent systems are supported by agent platforms that basically provide agent programming libraries and system-level services to support agent execution. An important system-level feature that an agent platform needs to include for the development of robust and fault-tolerant agent systems is agent state execution capture. In fact, agent checkpointing, persistency and strong mobility are mechanisms fully enabled by agent state execution capture. Agent checkpointing [13] is a technique for inserting fault tolerance into agent systems. In particular, it basically consists of storing a snapshot of the agent execution state and, later on, using it for restarting the agent execution in case of its failure. Agent persistency is a mechanism allowing saving an agent along with its state into the mass memory as a file. Archived agents can be later on loaded and resumed. Strong mobility [10] is an important feature of an agent that enables the migration of the agent state, data and code. Such mechanisms constitute the basis for supporting more complex, system-wide agent policies and algorithms for fault tolerance, load balancing, and transparent migration.

Unfortunately, even after more than one decade of research on agent systems, the majority of the currently available agent platforms, particularly those based on the standard Java Virtual Machine (JVM) do not provide this important feature at system-level. In fact, currently the standard JVM does not yet provide mechanisms to capture the state execution of Java processes. Nevertheless, several approaches have been to date proposed to overcome such an issue. They can be roughly taxonomized into three categories: system-level [4, 5, 14, 17], converters [3, 16, 15, 11, 12] and model-based [18, 9, 19]. The system-level approach modifies the underlying virtual machine to capture the execution state at process/thread level; however, the modified virtual machine is usually not compatible for agents previously developed. Conversely, converters and model-based approaches are application-level approaches that rely on specific, sometimes sophisticated, agent programming models and/or on converters at bytecode-level or source-code-level to allow for agent execution state capture. Although they do not require any virtual machine modification, the grain of capture of the agent execution state is usually coarse so that capture can be carried out only at specific points of agent execution. In fact, capture is either automatically driven by specific statements (in agent converters) or manually programmed by exploiting the reference agent programming models. Both techniques can be used only for agent-driven capture and not for an effective system-driven capture.

In this paper, an application-level technique for agent-driven and system-driven capture of agent state execution is proposed. Agent-driven capture is based on three key statements (checkpoint, persistence and move), whereas system-

* Corresponding author

driven capture is fine-grain and carried out on a per-instruction basis. A distinctive feature of the defined technique is the capture of the execution state in recursive and mutually recursive methods that can also contain key statements. The proposed technique is based on the Distilled StateCharts Star (DSC*) formalism that provides an agent-oriented type of recursive hierarchical state machine through which the agent program and its execution state can be represented. According to the technique, a single-threaded agent program formalized as a main method and ancillary methods is translated into a DSC* object, preserving the original agent program semantics. The technique is currently implemented in Java and integrated into the JADE framework [2]. In particular, an agent program, which is formalized through a simple Java-based Agent Language (JAL), is converted into a JADE agent compliant to a newly defined JADE behavior named DSCStarBehaviour. Conversion can be carried out in two modes: (i) driven by the statements *checkpoint*, *persistence* and *move*; (ii) instruction by instruction. The so obtained JADE behavior not only has the same semantics as the original agent program but also incorporates its execution state at run-time. A simple yet effective example, concerning an agent-based version of the Fibonacci program, is used to exemplify the proposed technique.

The rest of this paper is organized as follows. In Section II the DSC* formalism is defined as an enhancement of the Distilled StateCharts formalism [9]. Section III presents the proposed technique that is then exemplified in section IV by means of the Fibonacci agent example. Section V provides some details about the implementation of the proposed technique through Java and the JADE framework. Finally, conclusions are drawn and future work anticipated.

II. The Distilled StateCharts Star Formalism

The Distilled StateCharts Star (DSC*) formalism is based on of the Distilled StateCharts (DSC) formalism [9, 8] that was specifically defined for effective modeling of single-threaded agent behavior through hierarchical state machines based on ECA rules, OR-decomposition, history entrance mechanisms and UML-like execution semantics based on the run-to-completion step. In addition, DSC* introduce typical mechanisms of recursive functions/procedures so that DSC* machines are recursive hierarchical state machines [1]. A DSC* machine is formalized by the following tuple:

DSC* = <Σ, s_0, *fs*, *L*, Φ, *deep*, *defaultHistoryEntrance*, *defaultEntrance*, *defaultEntranceAction*, *V*, *star*, *return*>, where:

- Σ is the set of states including composite states (cs=ncs$\cup$pcs), which can be normal composite states (ncs) or procedure composite states (pcs), simple states (ss) and history pseudostates (hs). State notation is defined according to the hierarchical structure of the DSC*. In particular: (i) the notation A(B) indicates that the simple state B is encapsulated in the composite state A; (ii) the notation A(•) indicates that A is a composite state; (iii) the notation A(x), where x is a placeholder variable, indicates any state included in A (e.g. x=B).
- s_0 is the initial state.
- *fs* is the set of final states.
- *L* is the set of transition labels. A label is defined as the following ECA rule: *Event[Condition]/Action*. The semantics implies that if Event is processed and Condition is true, upon transition firing Action is first atomically executed and, then, the state transition takes place.
- Φ is the set of transitions among states. In particular, it is defined as follows: $\Phi \subseteq \Sigma \times \Sigma \times L$. Thus a transition is formalized as a triple <source state, target state, label>.
- *defaultEntrance* is a function that indicates the default entrance of each composite state. The default entrance is the transition sourcing from the initial pseudostate of a composite state and targeting a state encapsulated into the composite state. Default entrances are not included in Φ as initial pseudostates are implicitly associated to composite states and are not explicitly defined in Σ.
- *defaultEntranceAction* is a function that associates an action (if any) to the default entrance of composite states.
- *deep* is an attribute of a history pseudostate indicating deep history if true, shallow history if false. History mechanisms allow a partial (through shallow history pseudostate H) or full (through deep history pseudostate H*) recovery of the state history after re-entering into a composite state previously exited.
- *defaultHistoryEntrance* is a function that indicates the default history entrance (if any) of the history pseudostates. The default history points to the state to be entered in case history is not initialized, i.e. when the composite state containing the history pseudostate is entered for the first time.
- *V* is a set of variables hierarchically scoped according to the state structure. In particular, $V(x)=\{v_1(x),\ldots,v_n(x)\}$ is the set of variables declared in the x state.
- *star* and *return* are attributes of specific transitions which respectively support activation of and return from a procedure type composite state (or procedure state) that semantically behaves like a procedure in an imperative-style programming language. Thus the *star* transition represents the procedure call whereas the *return* transition represents the procedure return. Let A and B be two composite states, when the transition (*) from A to B is drawn, the transition (<<R>>>) from B to A is to be added too (see Figure 1). The CallB event is generated inside A and triggers the (*) transition, whereas the ReturnFromB event is generated inside B and triggers the (<<R>>) transition. Procedure call parameters are passed to B through the CPE_B event that is generated inside A just after the CallB generation. Return value is passed through the RPE_B event that is generated just after the ReturnFromB event. The procedure state activation and return semantics are as follows: if the DSC* is in the A state and the CallB event is processed, the transition (*) is fired so that a new instance of the B

state is entered and the B state parameters (if any) are actualized with the parameter values contained in the CPE_B event. When the B state has to complete, the ReturnFromB event and the RPE_B event, filled with the return value, are generated. Once the ReturnFromB event is processed by the DSC*, the transition <<R>> fires so that the B state instance is removed and the A state is entered through its shallow history pseudostate. The so defined mechanism is therefore equivalent to the procedure call mechanism; parameters are only passed by value. Procedure state recursion and mutual recursion among procedure states are also fully supported (see Section III).

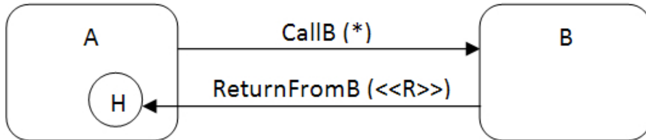


Fig. 1 Star (*) and return (<<R>>) transitions

The structure of a well-formed DSC* consists of a top state enclosing one procedure state, named main procedure state, which contains the initial state and the main final state, and zero or more secondary procedure states linked directly or indirectly to the main procedure state (see Section III for an example).

The execution semantics of a DSC* machine are defined in terms of an abstract machine, which embodies a DSC*, whose key components are:

- An event queue (EQ), which holds incoming event instances until they are dispatched;
- An event dispatching mechanism (EDM), which selects and de-queues event instances from EQ.
- An event processor (EP), which processes dispatched events.

The semantics of event processing is based on the run-to-completion (RTC) assumption implying that an event can only be de-queued and dispatched if the processing of the previous event is fully completed.

Given the last dequeued event by EDM, the main task of EP is to cyclically:

1. find out which transitions are enabled, i.e., which transitions could be fired based on the dequeued event;
2. select, among the enabled transitions, the transitions to be fired. Indeed, only one transition of a DSC* is step-by-step ready to be fired.

The algorithm executed by EP is reported in Figure 2 through a pseudocode notation.

The input to each step is given by the last recently dequeued event (`current_event`) and the DSC* execution control status, which includes the current state (`current_state`) of the DSC*, the dynamic history information associated to history pseudostates (history(hs_i), $\forall i$), and the stacks of the procedure states (stack(cs_j), $\forall j$). At the termination of each step, the output is given by the new DSC* execution control status.

```
1 :for each history pseudostate i
2 : history[i].init(defaultHistoryEntrance[i])
3 :for each procedure state j
4 : stack(j).init(null)
5 :current_state = s0;
6 :while (current_state∉fs) do
7 : current_event = EQ.dequeue();
8 : enabled_transitions =
     enabled(current_state, current_event);
9 : if (enabled_transitions∉∅) then
10:  fired_transition = fire(enabled_transitions);
11:  if (return(fired_transition)) then
12:   pop(stack(source(fired_transition)));
13:  else
14:   if (source(fired_transition)∈cs)
15:    updateHistory(source(fired_transition))
16:   endIf
17:  endIf
18:  executeActionChain(fired_transition);
19:  if (star(fired_transition)) then
20:   j=target(fired_transition);
21:   push(stack[j], new instance of j);
22:  endIf
23:  current_state = nextconf(current_state,
                              fired_transition);
24: endIf
25:endWhile
```

Fig. 2 The DSC* execution semantics algorithm

Before starting the event processing loop, history pseudostates are initialized with the default history entrances, the stack of the procedure states is set to null and the `current_state` set to the initial state (lines 1-5). While the `current_state` is different from a final state (line 6), the `current_event` is dequeued (line 7) and successively processed. In particular the following steps are carried out: (i) the set of enabled transitions is computed on the basis of the `current_state` and `current_event` (line 8); (ii) if such a set is not empty the transition to be fired is selected among the enabled transitions (lines 9-10); (iii) if the fired transition is a <<R>> transition then the instance at the top of the stack of the procedure state which is the source of the transition is removed (lines 11-12); otherwise, if the source state of the fired transition is composite, its history pseudostates are updated (lines 14-15); (iv) the action chain of the fired transition is executed (line 18); (v) if the fired transition is a (*) transition a new instance of the target procedure state is added to its stack (lines 19-22); (vi) finally the new current state is computed on the basis of the `current_state` and the fired transition (line 23).

III. A DSC*-based Technique for Agent State Execution Capture at Application-level

The proposed technique translates an agent program, defined according to the schema reported in Figure 3, into a DSC* machine that is semantically equivalent to the agent program and contains its execution state at run-time. Translation can be carried out either per instruction basis (named full translation) or driven by key statements. The agent program, which represents a single-threaded agent, is composed of a main method (simply called main) from which the agent

activity starts and zero, one or more supporting methods $\{m_1,..., m_N\}$, N>=0. Agent data can be declared at global level (GV) and at local level in each method (LVm_i). The supported control-flow statements are: sequential, selective (if-then-else), iterative (for and while), procedure call and return. Moreover the defined key statements are: (i) *checkpoint*, which captures the execution state of the agent and store it in memory; (ii) *persistence*, which captures the execution state of the agent and store it in a file; and (iii) *move*, which triggers the migration of the agent from the current location to a new location.

```
program Agent{
 GV //Declaration of Global Variables
 m1(param1,…,paramn_m1):return_value_m1{
  LV1 // Declaration of Local Variables
  CBm1 //code block of the method 1
 }
 …
 mN(param1,…,paramn_mN):return_value_mN{
  LVN // Declaration of Local Variables
  CBmN //code block of the method N
 }
 main(param1,…,paramn_main){
  LV_main // Declaration of Local Variables
  CBm_main //code block of the main
 }
}
```

Fig. 3 The schema of the Agent program

The structure of the Agent program is represented by the equivalent DSC* machine structure shown in Fig. 4. Each procedure is formalized as a procedure state, the global variables V(Active) are declared at the top state named Active, which is a normal composite state, the local variables of the main V(main) along with the main parameters P(main) at the main level, the local variables $V(m_i)$ of the supporting methods along with the method parameters $P(m_i)$ at the level of the corresponding procedure states. The Active composite state, which is embedded in the FIPA agent behavior template [6], is always entered through the deep history pseudostate to allow restoring the DSC* machine status to the status in which the DSC* machine was before leaving the Active state.

Each statement of the Agent program can be translated into a DSC* diagram. In the following we describe the translation patterns of the Agent program statements and, finally, we present a short version of the translation algorithm. We defined two types of statements: special statements, which include control-flow statements (if-then-else, while, for, procedure call, return) and key statements (checkpoint, persistence, move), and normal statements, which include all other statements different from the special statements.

In Figure 5 the translation of the normal statement is reported. The E event drives the execution of the `statement` bringing the DSC* machine into the S state which formalizes the control-flow status just after the execution of the `statement`. A new event E is generated to drive the control-flow to the next control-flow point. This translation is carried out if and only if the translation is full.

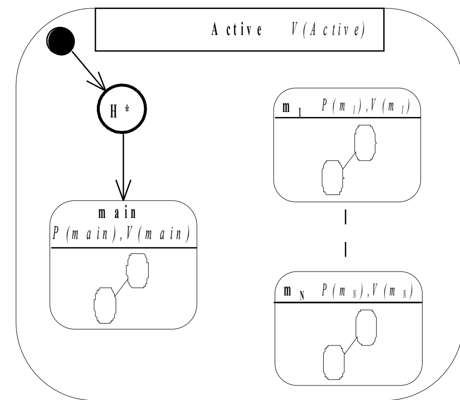


Fig. 4 The schema of DSC* machine of an Agent program

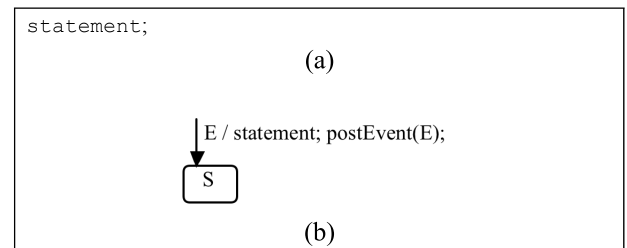


Fig. 5 Translation of the normal statement: (a) normal statement; (b) corresponding DSC* diagram

The translation of the `key_statement` is shown in Figure 6. In this case, the list of normal statements (containing one or more normal statements) which are before the `key_statement` are moved into the transition action before the key statement. This translation is carried out if and only if the translation is key-statement-driven; otherwise the translation is carried out as for a normal statement (see above).

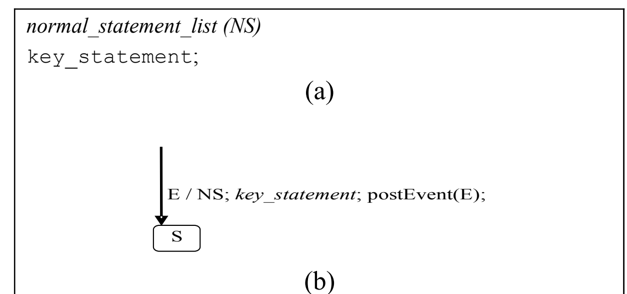


Fig. 6 Translation of the key statement: (a) key statement; (b) corresponding DSC* diagram

The translation of the `if-then-else` statement is reported in Figure 7. The BEFORE_IF state indicates the control flow point before the execution of the `if-then-else` statement. COND_TRUE and COND_FALSE represent the states that indicate the points just after the `CONDITION` evaluation and before the execution of the `IF_BLOCK` and the `ELSE_BLOCK`, respectively. The AFTER_IF and AFTER_ELSE states represent the points after the execution of the `IF_BLOCK` and `ELSE_BLOCK`, respectively. Finally the END state represents the point after the `if-then-else` statement.

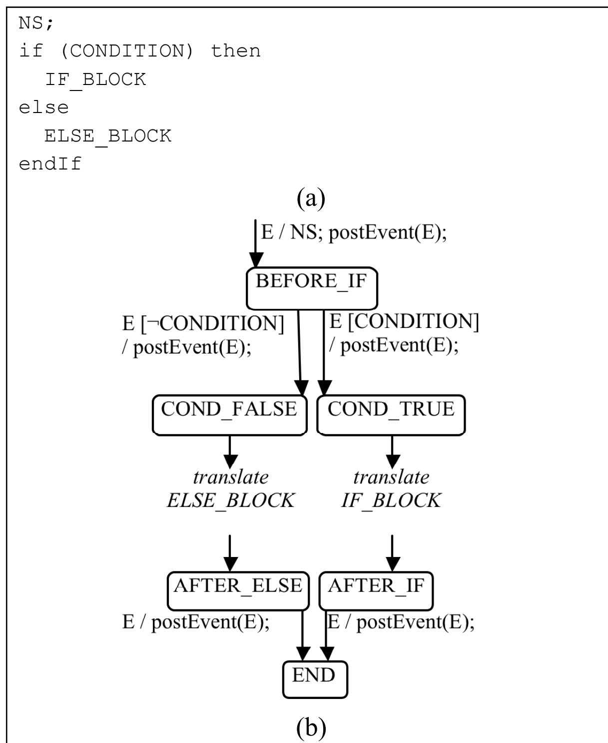


Fig. 7 Translation of the if-then-else statement: (a) if-then-else statement; (b) corresponding DSC* diagram

The translation of the `while` statement is reported in Figure 8. The Loop state indicates the control-flow point before the execution of the `while` statement. The Cond_True state represents the state indicating the point just after the `CONDITION` is evaluated true and before the execution of the `WHILE_BLOCK`. If the `CONDITION` does not hold the Cond_False state is reached which represents the control flow point after the `while` statement.

Figure 9 reports the translation of the `for` statement. The Loop state indicates the control-flow point before the execution of the `for` statement. The `for` counter `i` is initialized in the action of the transition above Loop. The Cond_True state represents the state indicating the point just after the `condition i<M` is evaluated true and before the execution of the `FOR_BLOCK`. If `condition` does not hold the Cond_False state is reached which represents the control-flow point after the `for` statement. The `for` counter `i` is incremented in the action of the transition from Continue to Loop. Of course, the `for` condition as well as the `for` counters can be easily generalized.

The translation of the `procedure call` is reported in Figure 10. To implement the call to the P2 method, the CallP2 and CPE_P2 events are sequentially posted. The former labels the (*) transition between P1 and P2. The latter contains the parameter values {PARAMETERS_P2} to be passed to P2. When the CPE_P2 event is processed, the parameters value are extracted and assigned. When the RPE_P2, generated in the P2 state (see translation of the `return` statement), is processed, the return value contained in the RPE_P2 event is extracted and assigned.

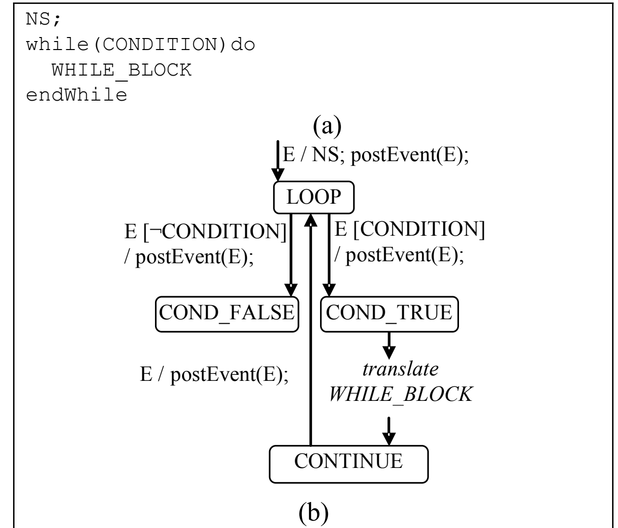


Fig. 8 Translation of the while statement: (a) while statement; (b) corresponding DSC* diagram

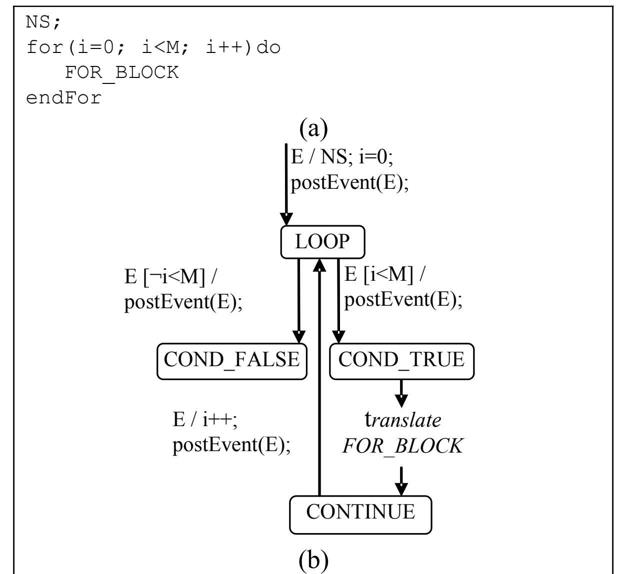


Fig. 9 Translation of the for statement: (a) for statement; (b) corresponding DSC* diagram

The translation of the `return` statement is reported in Figure 11. The ReturnFromP2 and RPE_P2 events are sequentially posted. The former drives the <<R>> transition between P2 and P1. The latter contains the return parameter value to be passed back to P1.

The translator algorithm (Figure 12) converts an agent program (AP) defined as in Figure 3 into a DSC* diagram (AP*), preserving the AP semantics. In particular, for each method m of AP (line 1) the following steps are carried out:

(i) a procedure state pcs corresponding to m is created and added to AP* (line 2);

(ii) the variables P(m) and V(m) are added to pcs (line 3);

(iii) the initial state of pcs is defined along with the transition labeled by the CPE event, through which the method parameters could be passed, which connects s1 and s2 repre-

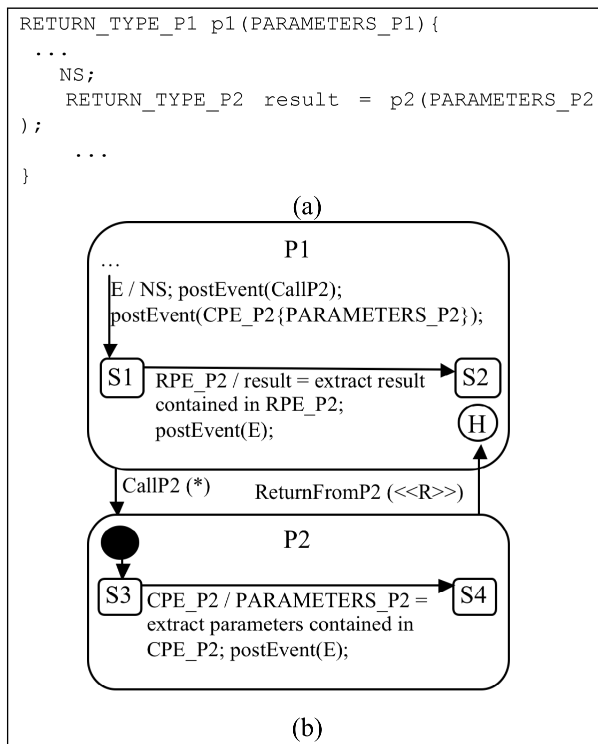


Fig. 10 Translation of the procedure call statement: (a) procedure call statement; (b) corresponding DSC* diagram

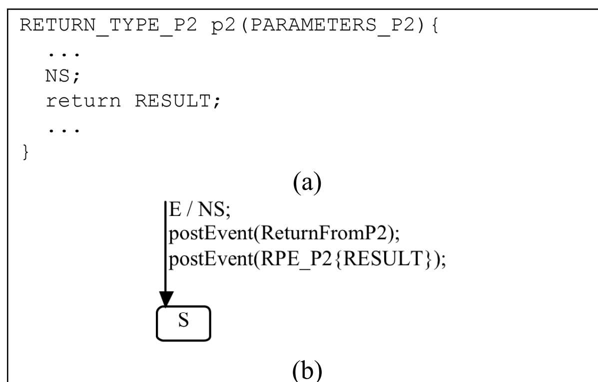


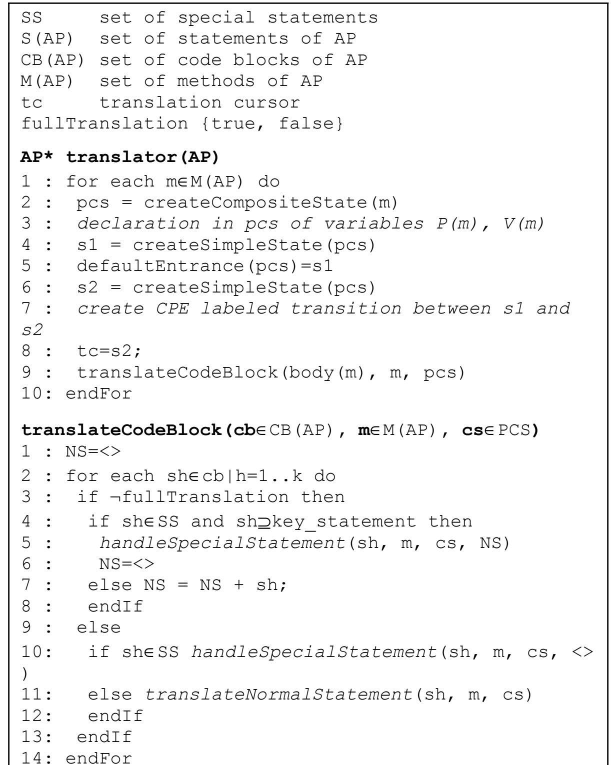
Fig. 11 Translation of the return statement: (a) return statement; (b) corresponding DSC* diagram

senting the control-flow starting point of m (lines 4-7); the tc variable represents the translation cursor pointing to the last created state from which translation is to be restarted;

(iv) the body of m is finally translated through translate-CodeBlock. In particular, translateCodeBlock sequentially translates the statements belonging to the code block cb from the 1st to the k-th. Given the h-th statement (s_h) the following steps are carried out:

(i) if `full_translation` is false and s_h belongs to SS and contains directly or indirectly a key_statement then s_h is translated according to the translation patterns and NS is reset (lines 3-6); otherwise, s_h is inserted in NS (line 7);

(ii) if `full_translation` is true and s_h belongs to SS then s_h is translated according to the above described transla-



```
SS      set of special statements
S(AP)   set of statements of AP
CB(AP)  set of code blocks of AP
M(AP)   set of methods of AP
tc      translation cursor
fullTranslation {true, false}

AP* translator(AP)
1 : for each m∈M(AP) do
2 :  pcs = createCompositeState(m)
3 :  declaration in pcs of variables P(m), V(m)
4 :  s1 = createSimpleState(pcs)
5 :  defaultEntrance(pcs)=s1
6 :  s2 = createSimpleState(pcs)
7 :  create CPE labeled transition between s1 and s2
8 :  tc=s2;
9 :  translateCodeBlock(body(m), m, pcs)
10: endFor

translateCodeBlock(cb∈CB(AP), m∈M(AP), cs∈PCS)
1 : NS=<>
2 : for each sh∈cb|h=1..k do
3 :  if ¬fullTranslation then
4 :   if sh∈SS and sh⊒key_statement then
5 :    handleSpecialStatement(sh, m, cs, NS)
6 :    NS=<>
7 :   else NS = NS + sh;
8 :   endIf
9 :  else
10:   if sh∈SS handleSpecialStatement(sh, m, cs, <>)
11:   else translateNormalStatement(sh, m, cs)
12:   endIf
13:  endIf
14: endFor
```

Fig. 12 A schema of the translation algorithm

tion patterns (line 10); otherwise, s_h is translated as normal statement (line 11).

IV. A Translation Example: Fibonacci Agent

To exemplify the technique presented in the previous section, the translation of a Fibonacci mobile agent, reported in Figure 13, is described. The agent is an extension of the Fibonacci algorithm with mobility: it moves across N locations to compute fibonacci(N) and, finally, prints out the result at the home location.

The translation was carried out both full and statement-driven; for the sake of exemplification, in the following we discuss the DSC* diagram of the Fibonacci agent obtained according to the key-statement-driven translation (see Figure 14). In particular, the `move` statement drives the translation so that the agent execution state can be captured before any migration and restored after migration. It is worth pointing out the DSC*-based modeling of the direct recursion: the Fibonacci procedure state has both a self (*) transition labeled by the CallFibonacci event and a self <<R>> transition labeled by the ReturnFromFibonacci event.

V. A JADE-based Implementation

The proposed technique is implemented in Java for the JADE framework that is one of the most diffused agent platforms [2]. The implementation comes with a translator program and a JADE add-on, named DSCStarBehaviour framework, which provides the programming abstractions supporting the definition of agent behaviors in terms of DSC* ma-

```
public class FibonacciAgent extends JALAgent {
 Location [] locations;
 public int fibonacci(int n){
  int x;
  int res1;
  int y;
  int res2;
  int res;
  if (n == 0 || n == 1) {
   move(locations[0]);
   return n;
  }
  move(locations[n]);
  x = n - 1;
  res1 = fibonacci(x);
  y = n - 2;
  res2 = fibonacci(y);
  res = res1 + res2;
  return res;
 }
 public void main(Location [] loc){
  int result;
  locations = loc;
  result = fibonacci(10);
  System.out.println("RESULT = " + result);
 }
}
```

Fig. 13 Fibonacci agent

chines. In particular, a Java-based agent program is converted by the translator into a JADE agent based on the DSCStarBehaviour. The agent program is defined through a Java-based agent language (JAL) which allows structuring agents as single-threaded Java programs using classes of the JADE framework and the three key statements (checkpoint, persistence, and move). The DSCStarBehaviour framework is an enhancement of the JADE DistilledStateChartBehaviour framework [7]; a simplified version of its class diagram is reported in Figure 15.

The DSCStarEvent extends ACLMessage so that the basic message processing mechanisms of JADE are completely reused. The DSCStarBehaviour extends CompositeBehaviour and implements mechanisms fulfilling the DSC* execution semantics. All DSC* composite states extend DSCStarBehaviour and are handled through the JADE Behaviour mechanisms. The other classes are abstractions of the main components of the DSC* formalism: variables and parameters of procedure states (DSCStarVariablesAndParameters), (*) transition (DSCStarStarTransition) and <<R>> transition (DSCStarReturnTransition) which are an extension of normal transition (DSCStarNormalTransition), and instances of a procedure state (VirtualInstance).

A simplified excerpt of the obtained FibonacciAgent class is reported in Fig 16 to provide a flavor of programming with the JADE DSCStarBehaviour framework. `FibonacciAgent` extends the basic JADE Agent class and contains the two procedure states `main` and `fibonacci` (lines 1-3). In the `setup` method all the simple states are defined along with the transitions; finally the `FibonacciAgent` behavior is started up (lines 4-28). In particular: (i) states `S1`-`S17` are declared, created and added to `main` and `fibonacci` as shown in lines 7-9; (ii) in line 10, the `root` behavior, which represents the encapsulating Active state (see Fig. 14), is created; (iii) the recursive (*) transition is defined in line 11; (iv) the transition `t2` between the `S12` and `S13` states (see Fig. 14) is defined in lines 12-24; two methods are defined for each transition: `trigger`, which defines the transition trigger *event[condition]*, and `action`, which defines the transition action, e.g. action of `t2`; (v) all transitions are added to the `root` which is added to the pool of active behaviors (lines 25-26); (vi) finally, the CPE_MAIN event, which triggers the execution of the FibonacciAgent, is created and posted (lines 27-28).

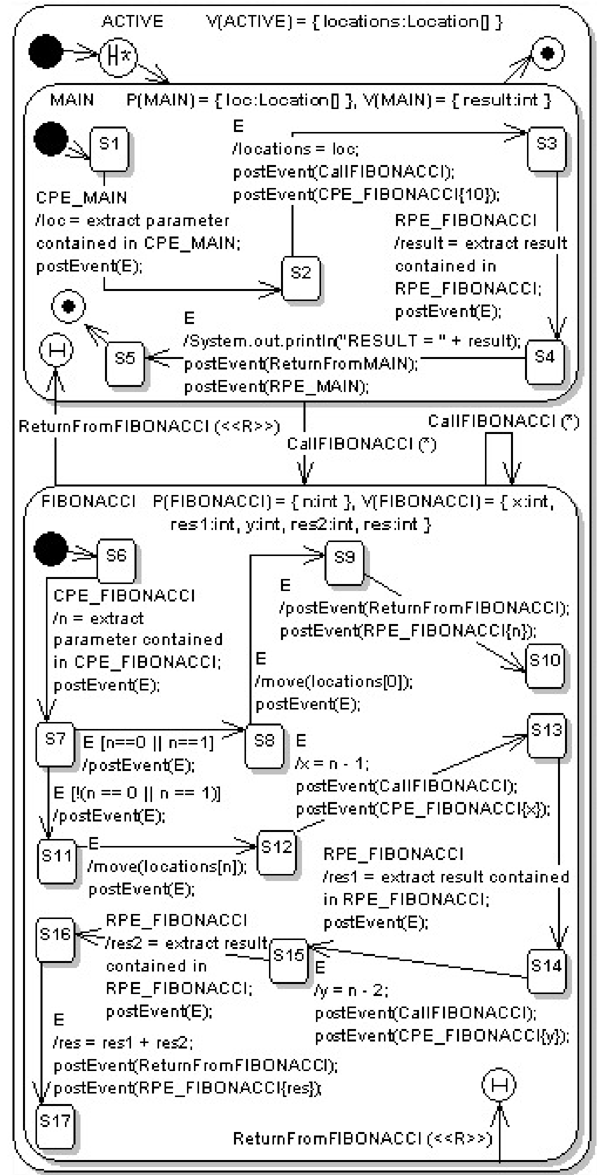


Fig. 14 DSC* diagram of the Fibonacci agent based on the *move-driven* translation

VI. Conclusion

In this paper we have presented a novel technique based on hierarchical recursive state machines for application-level capture of agent execution state. In particular, the technique uses the DSC* formalism to formalize agent programs writ-

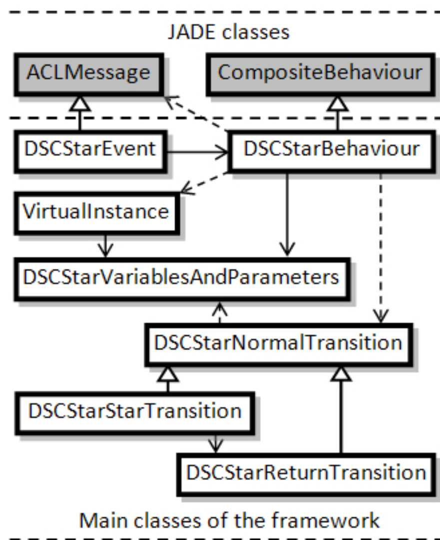


Fig. 15 The DSCStarBehaviour class diagram

ten in an imperative style language. The technique is currently implemented in Java for the JADE framework and allows obtaining JADE agents, whose behavior is based on the newly defined DSCStarBehaviour framework, which can be actively and passively checkpointed, made persistent and strongly migrated. On-going research efforts are devoted to (i) analyze the code and execution overhead introduced by the proposed application-level approach and (ii) implement the technique for other Java-based agent platforms.

```
1:public class FibonacciAgent extends Agent {
2: private DSCStarBehaviour fibonacci;
3: private DSCStarBehaviour main;
4: protected void setup(){
5:  fibonacci=new DSCStarBehaviour(this,"FIBONACCI",..);
6:  main=new DSCStarBehaviour(this,"MAIN",...);
7:  Behaviour S1=new SimpleStateBehaviour(this,"S1");…;
8:  main.addInitialState(S1); main.addState(S2);…;
9:  fibonacci.addInitialState(S6);
    fibonacci.addState(S7);…;
10: DSCStarBehaviour root =
       DSCStarBehaviour.createRootForDSCStar(main,...);
11: DSCStarStarTransition t1 = new
            DSCStarStarTransition(fibonacci, fibonacci);
12: DSCStarNormalTransition t2 = new
                 DSCStarNormalTransition("T2",S12,S13){
13:  public boolean trigger(ACLMessage msg,...) {
14:   if (msg instanceof E) return true;
15:   return false;
16:  }
17:  public void action(...) {
18:   VarFibonacci varFibonacci = (VarFibonacci)
                         sourceVariablesAndParameters;
19:   varFibonacci.x = varFibonacci.n - 1;
20:   DSCStarEvent call = new
              DSCStarEvent(DSCStarEvent.CALL,fibonacci);
21:   postMessage(call);
22:   CPEFibonacciEvent cpe = new
          CPEFibonacciEvent(DSCStarEvent.CPE,fibonacci);
23:   cpe.n = varFibonacci.x; postMessage(cpe);
24:  }}
25: root.addTransition(t1);…;root.addTransition(t17);
26: addBehaviour(root);
27: CPEMainEvent cpe_main = new
              CPEMainEvent(DSCStarEvent.CPE,main);
28: postMessage(cpe_main);}}
```

Fig. 16 A FibonacciAgent code excerpt

References

R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis, “Analysis of recursive state machines,” *ACM Transactions on Programming Languages and Systems*, 27(4), 2005.

F. Bellifemine, A. Poggi, and G. Rimassa,, “Developing multi agent systems with a FIPA-compliant agent framework,”. *Software Practice And Experience* 31, 103-128, 2001.

L. Bettini and R. De Nicola, “Translating Strong Mobility into Weak Mobility,” *Proc. of 5th IEEE Conference on Mobile Agents*, G. Picco (ed), LNCS 2240, 2001, pp. 182-197.

S. Bouchenak and D. Hagimont, “Pickling threads state in the Java system,” *Proc. of Technology of Object-Oriented Languages and Systems Europe – Europe* (TOOLS Europe’2000), Mont Saint Michel / Saint Malo, France, Jun. 2000.

S. Bouchenak and D. Hagimont, “Zero Overhead Java Thread Migration,” *Technical Report N°0261*, INRIA, Montbonnet-St-Martin(France), May 2002.

FIPA Agent Management Specification, Management for agents on FIPA agent platforms, http://www.fipa.org/specs/fipa00023/SC00023K.html.

G. Fortino, F. Rango, W. Russo, “Statecharts-based JADE agents and tools for engineering Multi-Agent Systems”, *Proc of 14th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems* (KES2010), Cardiff, 2010.

G. Fortino, A. Garro, S. Mascillaro, W. Russo, “Using Event-driven Lightweight DSC-based Agents for MAS Modeling,” *International Journal on Agent Oriented Software Engineering*, 4(2), 2010.

G. Fortino, W. Russo, E. Zimeo, “A statecharts-based software development process for mobile agents,” *Information and Software Technology* 46(13), 907--921, 2004.

A. Fuggetta, G.P. Picco, and G. Vigna, “Understanding Code Mobility”, *IEEE Trans. on Software Engineering*, 24(5), pp. 342-361, 1998.

S. Funfrocken, “Transparent Migration of Java-based Mobile Agents: capturing and re-establishing the state of Java programs,” *Proc. of the 2nd Int’l Workshop on Mobile Agents*, K. Rothermel and F. Hohl (eds), LNCS 1477, pp. 26-37, Sept. 1998, pp. 26-37.

M. Hohlfeld and B. Yee, “How to migrate agents,” available at http://www.cs.ucsd.edu/~bsy, 1998.

Y. Ji, H. Jiang, and V. Chaudhary,. “Adaptation point analysis for computation migration/checkpointing,” *Proc. of the 2005 ACM Symposium on Applied Computing*, Santa Fe, NM, Mar 13 - 17, 2005.

R. Quitadamo, L. Leonardi, G. Cabri, “Leveraging strong agent mobility for Aglets with the Mobile JikesRVM framework”, *Scalable Computing: Practice and Experience*, 7 (4), December 2006.

T. Sakamoto, T. Sekiguchi, and A. Yonezawa, “Bytecode Transformation for Portable Thread Migration in Java,” *Proc. of the 4th Int’l Symp on Mobile Agents*, Zurich, Switzerland, Sept. 2000.

T. Sekiguchi, H. Masuhara, and A. Yonezawa, “A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation,” Coordination Languages and Models, LNCS 1594, Apr. 1999.

N. Suri, J.M. Bradshaw, M.R. Breedy, P.T. Groth, G.A. Hill, and R. Jeffers, “Strong mobility and fine-grained resource control in Nomads,” *Proc. of the 4th Int’l Symp on Mobile Agents*, pp. 79-92, Zurich, Sept. 2000.

M. Zhang and W. Li, “Persisting Autonoumous Workflow for Mobile Agents Using Mobile Thread Programing Model,” LNAI 1733, pp. 84-93, 2002.

C. Wicke, L.F. Bic, M.B. Dillencourt, and M. Fukuda, “Automatic State Capture of self-migrating computations in MESSENGERS,” Proc. of the 2nd Int’l Workshop on Mobile Agents, LNCS 1477, Springer-Verlag, pp. 68-79, 1998.

M. Wooldridge, N. Jennings, “Intelligent agents: theory and practice,” The Knowledge Engineering Review, 10(2), 115-152, 1995.