An Automatic Code Generation Tool for JADE Agents

FLORIN STOICA
Computer Science Department
University "Lucian Blaga" Sibiu
Str. Dr. Ion Ratiu 5-7, 550012, Sibiu
ROMANIA

Abstract: - SDL (Specification and Description Language) is an object-oriented, formal language defined by The International Telecommunications Union – Telecommunications Standardization Sector (ITU-T), applicable to the specification and implementation of distributed systems. SDL is capable of describing the evolving state of asynchronous, concurrent systems, such as agent - based systems. The work presented in this paper consists on a tool responsible for generating JADE agents automatically from SDL specifications, in order to help the process of prototyping agent - based applications developed on JADE framework. The generated code is a completely functional Java code.

Key-Words: - SDL, JADE, agents, parser, FSM, ANTLR

1 Introduction

The key to successfully developing a system is to produce a thorough system specification and design. For this task, SDL is a suitable specification language. It is a graphical specification language that is both formal and object-oriented. The language is able to describe the structure, behaviour and data of real-time and distributed communicating systems with a mathematical rigor that eliminates ambiguities and guarantees system integrity. The most important characteristic of SDL is its formality. The semantics behind each symbol and concept are precisely defined. The specifications using SDL are intended to be formal in the sense that it is possible to analyse and interpret them unambiguously.
SDL is intended to specify the behavioural aspects of a system. Thus, a SDL specification of a system is the description of its required behaviour.
Agents are the fundamental specification concept of SDL-2000 [8]. The behaviour of an agent is described as an extended finite state machine: when started, an agent executes its start transition and enters the first state. The reception of a signal triggers a transition from one state to a next state. In a state, an agent may execute actions (tasks). Actions can assign values to variable attributes of the agent, branch on values of expressions, call procedures, create new agent instances and send signals to other agents.
In the following we present a model for SDL finite state machines, which is also suited for FSM-driven behaviour of a JADE agent, implemented by FSMBehaviour class. This model will help us to elaborate the mapping rules between SDL and JADE concepts, used by the SDL to JADE/Java code generation tool to translate an SDL specification into equivalent JADE code.

2 Reactive finite state machines

A reactive finite state machine is a tuple

$(Q, \boldsymbol{S}, \boldsymbol{D}, \boldsymbol{d}, q_0, F)$ where

Q is a finite, non-empty set of symbols called states,
$\boldsymbol{S}$ is a set of symbols representing valid inputs,
$\boldsymbol{D}$ is a set of symbols representing valid outputs,
δ is the state transition function: $\delta : Q \times (\Sigma \cup \{none\}) \rightarrow (Q \cup \{err\}) \times (\Delta \cup \{default\})$,
q_0 is an element of Q, the initial state,
F $\boldsymbol{Î}$ Q is the set of final states. `
Elements from $\boldsymbol{S}$ will be called *signals*, and elements from $\boldsymbol{D}$ will be called *events*. In one reaction, a FSM associate a current state p $\boldsymbol{Î}$ Q and an input signal a $\boldsymbol{Î}$ $\boldsymbol{S}$ with a next state $q$$\boldsymbol{Î}$ Q and an output event $b$$\boldsymbol{Î}$ $\boldsymbol{D}$, where $\boldsymbol{d}(p,a) = (q, b)$.
The behavior of an FSM is more easily understood when this is represented graphically in the form of a state transition diagram. The control states are represented by circles, and the transition rules are specified as directed edges. Each transition is labeled by event from $\boldsymbol{D}$ that triggers the transition. The arc without a source state denote then initial state of the system (state q_0).
During one reaction of the FSM, one transition is triggered, chosen from the set of admissible transitions (outgoing transitions from the current state), so that label of transition matches the terminating event of the current state. The FSM goes to the destination state of the triggered transition. Apparition of a terminating event for current state is conditionated by reception of one signal from $\boldsymbol{S}$ (leaving from a state could be done only if

was received a signal), an exception being the special signal *none*, which induce a *spontaneous transition*.

If terminating event of the current state $q \notin F$ is not explicit associated with an admissible transition, then:

- if exist the admissible transition labelled with *default*, this transition (called *implicit transition*) will be triggered;
- else FSM goes in an inconsistent state, denoted through *err*.

In case if FSM arrive in a state $q \in F$, after completeness of activities from that state, execution of finite state machine is stopped.

3 Jade agents with FSM behaviours

JADE is a middleware that facilitates the development of multi-agent systems and applications conforming to FIPA standards for intelligent agents [11].

The Agent class represents a common base class for user defined agents. The computational model of an agent is multitask, where tasks (or behaviours) are executed concurrently. A scheduler, internal to the base Agent class and hidden to the programmer, automatically manages the scheduling of behaviours.

A behaviour represents a task that an agent can carry out and is implemented as an object of a class that extends jade.core.behaviours.Behaviour. In order to make an agent execute the task implemented by a behaviour object it is sufficient to add the behaviour to the agent by means of the addBehaviour() method of the Agent class.

Each class extending Behaviour must implement the action() method, that actually defines the operations to be performed when the behaviour is in execution and the done() method (returns a boolean value), that specifies whether or not a behaviour has completed and have to be removed from the pool of behaviours an agent is carrying out. Scheduling of behaviours in an agent is not pre-emptive (as for Java threads) but cooperative. This means that when a behaviour is scheduled for execution its action() method is called and runs until it returns. The termination value of a behaviour is returned by his onEnd() method [2]. The path of execution of the agent thread is showed in the following pseudocode:

```
void AgentLifeCycle() {
 setup();
 while (true) {
  if (was called doDelete()) {
   takeDown();
   return;
  }
  Behaviour b =
  getNextActiveBehaviourFromSchedulingQueue();
  b. action();
  if (b.done() returns true) {
   removeBehaviourFromTheSchedulingQueue (b);
   int terminationValueOfTheBehaviour = b.onEnd();
  }
 }
}
```

Fig. 1 The life cycle of an JADE agent

Agent behaviours can be described as finite state machines, keeping their whole state in their instance variables.

The FSMBehaviour class provides the possibility of combining simple behaviours together (children) to create complex behaviours. The FSMBehaviour executes its children according to a Finite State Machine (FSM) defined by the user. More in details each child represents the activity to be performed within a state of the FSM and the user can define the transitions between the states of the FSM. When the child corresponding to state S_i completes, its termination value (as returned by the onEnd() method) is used to select the transition to fire and a new state S_j is reached. At next round the child corresponding to S_j will be executed. Some of the children of an FSMBehaviour can be registered as final states. The FSMBehaviour terminates after the completion of one of these children.

The following methods are needed in order to properly define a FSMBehaviour:

- public void **registerFirstState**(Behaviour state, java.lang.String name)

Is used to register a single Behaviour *state* as the initial state of the FSM with name *name*.

- public void **registerLastState**(Behaviour state, java.lang.String name)

Is called to register one or more Behaviours as final states of the FSM.

- public void **registerState**(Behaviour state, java.lang.String name)

Register one or more Behaviours as the intermediate states of the FSM.

- public void **registerTransition**(java.lang.String s1, java.lang.String s2, int event)

For the state s1 of the FSM, register the transition to the state s2, fired by terminating event of the state s1 (the value of terminating event is returned by onEnd() method, called when leaving the state s1 - sub-behaviour s1 has completed).

- public void **registerDefaultTransition**(java.lang.String s1, java.lang.String s2)

This method is useful in order to register a default transition from a state to another state independently on the termination event of the source state.

4 SDL Systems

SDL *Systems* consist of a structure of communicating *Agents*. Each agent may have variables, procedures, a state machine and a structure. An agent is characterised by the signals it may receive from and send to other agents, and by the procedures that it may perform upon request. SDL provides the following kinds of diagrams [12]: ***Agent diagrams*** that describe the properties of *Agents*, in terms of variables, procedures, an Agent state machine and contained Agents, ***State diagrams*** that depict the behaviour of Agents in terms of *States* and state *Transitions*, ***Procedure diagrams*** that depict the behaviour of *Procedures* and ***Package diagrams*** that define types that can be used in other diagrams.

The behaviour of an agent is described as an Extended Finite State Machine: when started, an agent executes its start transition and enters the first state. The reception of a signal triggers a transition from one state to a next state. In transitions, an agent may execute actions (tasks). Actions can assign values to variable attributes of the agent, branch on values of expressions, call procedures, create new agent instances and send signals to other processes. Communication by means of sending signals is asynchronous.

SDL gives a choice of two different syntactic forms to use when representing a system: a Graphic Representation (SDL/GR), and a textual Phrase Representation (SDL/PR). As both are concrete representations of the same SDL system, they are equivalent [5].

5 Agent behaviour specification in SDL

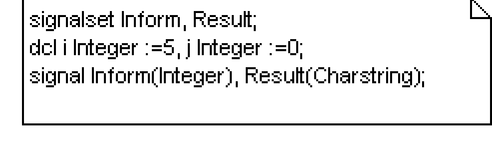
This section introduces a simple example for an agent behaviour specification in SDL [10]. For building SDL specifications, has been used Cinderella. Cinderella SDL is a CASE (Computer Aided Software Engineering) tool which is available from Cinderella (www.cinderella.dk).

Figure 3 shows the State diagram which describe behaviour of MyAgent agent as an Extended Finite State Machine (SDL/GR representation).

The SDL/PR representation of MyAgent is:

```
process MyAgent ;
  signalset Inform,Result;
  dcl i Integer :=5, j Integer :=0;
  signal Inform(Integer), Result(Charstring);
start;
task i:=i*2 ;
  nextstate  State_A ;
state State_B ;
    input none;
    output Result('i was greater') ;
    stop;
endstate;
state State_C ;
    input none;
    output Result('i was smaller') ;
    stop;
endstate;
  state State_A ;
    input Inform(j) ;
    task i:=i-j ;
    decision i>0 ;
      ( false ): nextstate  State_C ;
      ( true ):  nextstate  State_B ;
    enddecision;
  endstate;
endprocess;
```

Fig. 2 SDL/PR representation of MyAgent



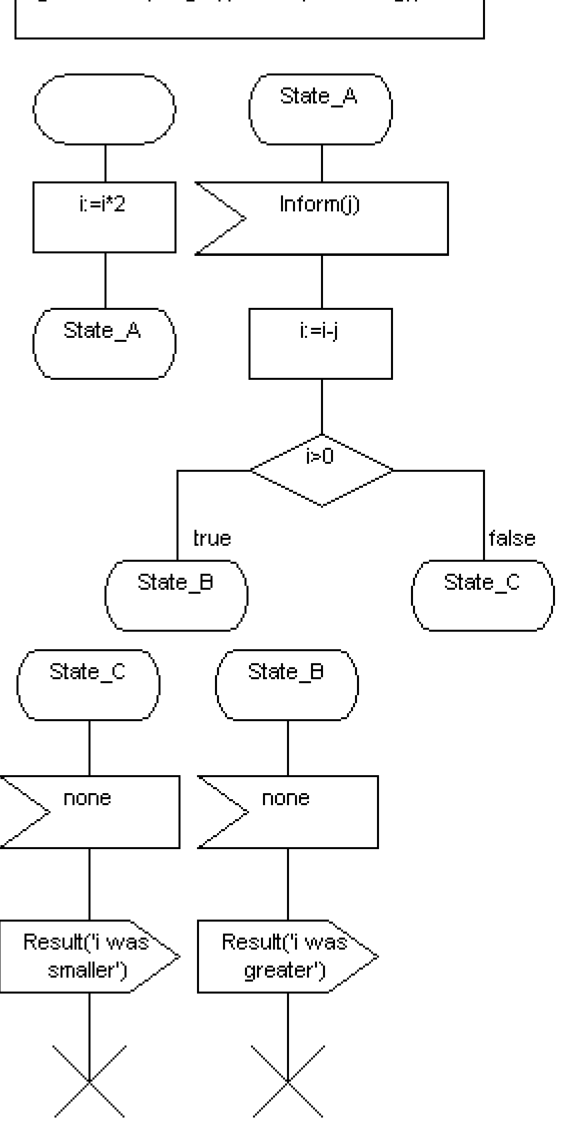


Fig. 3 SDL/GR representation of MyAgent

The following table contains explanations about symbols used in SDL/GR representation of MyAgent:

Symbol	Description
	A text symbol is a basic symbol which contains textual definitions (data types, signals etc.).
	The start symbol is used as start symbol for FSM
	The state symbol is used to represent a state of FSM
	The input symbol (receiving signals). A spontaneous signal symbol is an input symbol containing the text **none** (it induces spontaneous transitions)
	The task symbol (performing activities)
	The output symbol (sending signals)
	The decision symbol (branch execution)
	The stop symbol (execution of the finite state machine is stopped).

Table 1 SDL graphical symbols used in MyAgent specification

In Cinderella SDL, we can simulate our specification. When simulation is started, an environment process (Env) is created from which stimuli can be sent to the system and which holds in its input queue the signals sent from the system. If we are sending to MyAgent an *Inform* signal with parameter *9*, the agent replies with signal Result, having parameter 'i was greater'.

6 Generating JADE agents from SDL specifications

The work presented in this paper consists on a translator which generate automaticaly Java code parsing an SDL/PR specification that should have been developed in a previous step (an SDL specification defined in the graphical format SDL/GR can be saved in the textual format SDL/PR using the File export command from Cinderella).
In table 2 are defined mapping rules between SDL and JADE concepts. The SDL to JADE/Java code generation tool uses table 2 to convert the SDL specifications into equivalent JADE code.

SDL specification	FSMBehaviour of a Jade agent
Start symbol	The *setup()* method of the agent
State symbol	Child Behaviour registered as state of FSMBehaviour
Activities performed within a state	The action() method of child Behaviour associated with that state
Transition from current state to next state (*nextstate*)	The *done()* method of child Behaviour associated with current state returns *true*
Stop symbol	The doDelete() method of class Agent, called from child Behaviour associated with SDL state within is reached the *stop* symbol; this Behaviour will be registered as final state in FSMBehaviour
Signal receiving	Receiving a JADE message
Receiving a *none* signal in a certain state	Execution of activities from action() method of child Behaviour associated with that state (and generation of a terminating event for that state), unconditioned by reception of a proper signal

Table 2 Mapping rules between SDL and JADE

The tool reads a file that contains the SDL specifications in a textual form (SDL-PR) and, based on these specifications, produces the equivalent Java code for the behavioral description of the SDL system.
Typically, the procedure from requirements analysis to product implementation would involve the following steps:

- collect the initial requirements;
- make the SDL diagrams (specifications) to a level where they can be analysed, simulated and checked for consistency with the system requirements analysis (this can be done in Cinderella);
- when SDL design has proved consistent with the requirements, a code for the application can be generated.

In the following we present the tool responsible for generating JADE agents automatically from SDL design. This tool is based on the SDL parser developed by Michael Schmitt [6], which used ANTLR to build his own parser.

ANTLR, ANother Tool for Language Recognition, is a tool that accepts grammatical language descriptions and generates programs that recognize sentences in those languages. ANTLR knows how to build recognizers that apply grammatical structure to three different kinds of input: (i) character streams, (ii) token streams, and (iii) two-dimensional trees structures. Naturally these correspond to lexers, parsers, and tree walkers. The syntax for specifying these grammars, the *meta-language*, is nearly identical in all cases. ANTLR knows how to generate recognizers in Java, C++, C# [1].

The parser developed by Michael Schmitt reflects the SDL-2000 standard correctly and produces the abstract syntax tree (AST) of an SDL/PR specification. Specification of the SDL grammars is provided in three files: SDLLexer.g, SDLParser.g and SDLTreeParser.g; these correspond to generated lexer, parser, and tree walker, respectively. If the *buildAST* flag in the ANTLR parser options section is set to true, the parser created by ANTLR will read the source langauge and create ANTLR abstract syntax trees in memory. The tree walker undertake the complete building of the abstract syntax tree.

An abstract syntax tree (AST) captures the essential structure of the input in a tree form, while omitting unnecessary syntactic details. ASTs can be distinguished from concrete syntax trees by their omission of tree nodes to represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments. ASTs also omit tree nodes that represent unary productions in the grammar. Such information is directly represented in ASTs by the structure of the tree. Each node holds a token and pointers to its first child and next sibling:

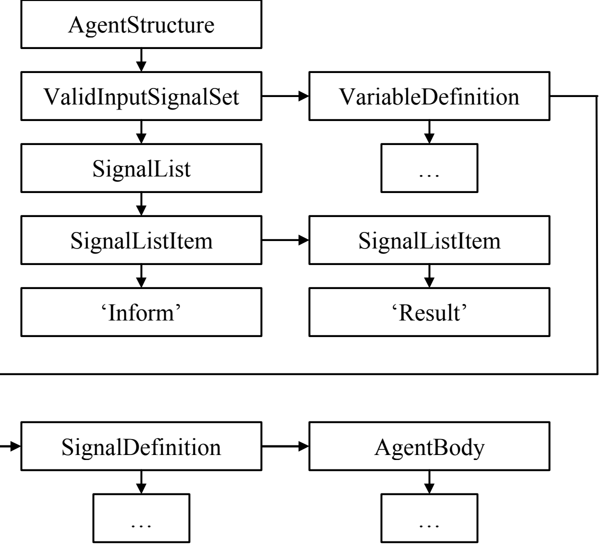


Fig. 4 A fragment from AST generated for the agent MyAgent described above

With the implementation of the AST components complete, the subsequent phases of the translator has been implemented. The code generator provide Java/JADE code walking a derived AST, completed with actions which are executed during tree walking. An action is a piece of code that run when a rule is matched. Actions can appear anywhere within a rule: before, during, or after a match.

The specification of the tree grammar is provided in the file SDLJADEParser.g. Computational tasks of the tree walker, generated by ANTLR from the tree grammar, are:

- completeness of building of the abstract syntax tree, initiated by the parser;
- walking the final AST;
- generation of Java/JADE code, through execution of actions associated with tree nodes, during tree walking.

The translator involve complex data structures. We decided to use the Standard Template Library (STL), which has C++ class templates for a wide range of basic building blocks of data structures, ranging from simple lists to associative arrays and hash tables.

For implementing the actions, the following variables/data structures were used:

```
static string firstState="";   //initial state
static string nextState="";    //next state
static string currentState=""; //current state
//associate each state with its current terminating code
//(event) which induce the transition;
// this represent the value of private variable _onEnd
//of the child behaviour which completes
typedef map<string, int, less<string>, allocator<int> >
TMapStateCode;
//associate each state of the FSMBehaviour with its child
//Behaviour
typedef map<string, string, less<string>,
allocator<string> > TMapStateBehaviour;
//associate each child Behaviour with its
//terminating condition from the done() method
typedef map<string, string, less<string>,
allocator<string> > TMapStateEnd;
//associate each variable with its type
//in order to translate correctly variable declarations
//and type conversions
typedef map<string, string, less<string>,
allocator<string> > TMapVarType;
// variable declarations
TMapStateCode mapStateCode;
TMapStateCode::iterator itStateCode; //iterator
TMapStateBehaviour mapStateBehaviour;
TMapStateBehaviour::iterator itStateBehaviour;
TMapStateEnd mapStateEnd;
TMapStateEnd::iterator itStateEnd; //iterator
```

```
TMapVarType mapVarType;
TMapVarType::iterator itVarType;  //iterator
```

Receiving a SDL signal is equated with receiving a JADE message:

```
stimulus {string msgName, msgVariable;}:
#( Stimulus
    s:signalListItem {msgName =
                          s->getFirstChild().get()->getText();}
(( v: variable { msgVariable =
       v->getFirstChild().get()->getText(); } | Void )+ )? )
{
 /* receiving a JADE message */
char * performative =
                       _strupr( _strdup( msgName.c_str() ) );
mapStateBehaviour[currentState].append(
"MessageTemplate mt = " +
"MessageTemplate.MatchPerformative(ACLMessage.");
mapStateBehaviour[currentState].append(performative);
mapStateBehaviour[currentState].append(");\n");
mapStateBehaviour[currentState].append(
  "ACLMessage msg = myAgent.receive(mt);\n");
mapStateBehaviour[currentState].append(
  "if (msg != null) {\n");
mapStateBehaviour[currentState].append(
  "String strVal = msg.getContent();\n");
/* get the variable type and make the corresponding
   conversion */
if ((itVarType = mapVarType.find(msgVariable)) = =
    mapVarType.end())
    cout << "Error! Variable: " << msgVariable <<
    " isn't  declared!" << endl;
else {
   /* only Integer type is treated here*/
      if ((*itVarType).second.compare("Integer") = =0) {
        /* conversion to integer value */
        mapStateBehaviour[currentState].append(
        msgVariable + " = Integer.parseInt(strVal);\n");
      }
      ...
}
mapStateCode[currentState]++;
mapStateBehaviour[currentState].append("_onEnd = " +
getStateCode(currentState) + ";\n");
mapStateBehaviour[currentState].append("}\n");
mapStateBehaviour[currentState].append("else {\n");
mapStateBehaviour[currentState].append("  block();\n");
mapStateBehaviour[currentState].append("}\n");
...
}  ;
```

After compilation, the generated agent MyAgent can be activated in the JADE platform. After receiving an INFORM message with content 9, the agent response is identical with that obtained from simulation of his specification ('i was greater'):

```
INFO: --------------------------------------
Agent container Container-2@JADE-IMTP://server2 is ready.
--------------------------------------------
```

i was greater

7 Conclusion

In this paper was presented implementation of a translator which generate automatically Java code parsing a SDL/PR specification of an agent, targeting the JADE platform. For this purpose, was ported on Windows platform (as a Microsoft Visual C++ 6.0 project) the SDL parser developed by Michael Schmitt on SuSE 8.1. This parser was extended with a tree parser grammar, SDLJADEParser.g. Because the project is based on ANTLR, was necessary to port on Microsoft Visual C++ the ANTLR runtime, resulting the library libantlr.lib, used to produce the generator executable.
In this stage of work, the generated code can be used as a prototype of a real agent-based application.

References:

[1] *ANTLR Reference manual*, http://www.antlr.org

[2] F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, *JADE programmer's guide*, http://jade.tilab.com

[3] G. Bucci, A. Fedeli, E. Vicario, Specification and Simulation of Real Time Concurrent Systems Using Standard SDL Tools, R. Reed (Ed.): *SDL 2003, LNCS 2708*, pp. 203–217, 2003.

[4] J. Floch, R. Braek, Using SDL for Modeling Behaviour Composition, R. Reed (Ed.): *SDL 2003, LNCS 2708*, pp. 36–54, 2003

[5] ITU-T Recommendation Z.100, *Specification and description language (SDL)*, International Telecommunication Union (ITU), 2000

[6] M. Schmitt, SDL-2000 parser/syntax checker, http://www.teststep.org/

[7] B. Moller-Pedersen, SDL Combined with UML, *Telektronikk 4.2000*

[8] R. Braek, A. Meisingset, The ITU-T Languages in a Nutshell, *Telektronikk 4.2000*

[9] SDL Forum Society, http://www.sdl-forum.org/

[10] F. Stoica, SDL executable specifications of agent-based systems, *The Proceedings of the International Economic Conference "25 Years of Higher Economic Education in Brasov"*, 2005, Editura Infomarket, ISBN 973-8204-72-0

[11] The Foundation for Intelligent Physical Agents (FIPA), www.fipa.org

[12] SINTEF ICT, TIMe: *The Integrated Method*, http://www.sintef.no/time/