Gaia Agents Implementation through Models Transformation

Nikolaos Spanoudakis[1,2] and Pavlos Moraitis[2]

[1]Technical University of Crete, Dept of Sciences, University Campus, 73100 Chania, Greece
`nikos@science.tuc.gr`
[2]Laboratory of Informatics Paris Descartes (LIPADE), Paris Descartes University,
45 rue des Saints-Pères, 75270 Paris Cedex 06, France
`{Nikolaos.Spanoudakis, pavlos}@mi.parisdescartes.fr`

Abstract. Gaia is a well-known Agent Oriented Software Engineering (AOSE) methodology. The emerging Model-Driven Engineering (MDE) paradigm encourages software modelers to automate the transition of one type of software model to another and eventually the code generation process. Towards this end we define a process for transforming the Gaia roles model liveness formulas to statecharts. This achievement on one hand allows the modeler to work on detailed agent design and permits, on the other hand, to automatically generate an agent's code using any one of the statecharts-based tools in the market.

Keywords: Agent Oriented Software Engineering, Statecharts, Gaia methodology, Model Driven Engineering

1   Introduction

During the last years, there has been a growth of interest in the potential of agent technology in the context of software engineering. A new trend in the Agent Oriented Software Engineering (AOSE) field is that of converging towards the Model-Driven Engineering (MDE) paradigm. Thus, a lot of well known AOSE methodologies propose methods and tools for automating models transformations in the meanwhile proposing metamodels in the modern ecore [1] or MOF [10] formats. Examples of such methodologies are Tropos [13] and Ingenias [3]. The Gaia methodology [19] is a popular methodology that, however, does not address the issue of transforming its design models to code. Efforts in the past have produced some results, however not in the MDE sense, that is without automating the process.

In this paper we present an automated process for transforming the Gaia roles model liveness property to a statechart [5]. The latter is a platform independent model (PIM) of the system to be, a result that is compatible with the Object Management Group (OMG) Model Driven Architecture (MDA) paradigm [7]. Moreover, the produced statechart is defined in a standardized format that can be used for defining new model to text transformations for any desired platform.

This process delivers several original results. The first result is the formal definition of the syntax of a Gaia liveness formula. Then, we define the statecharts [5] metamodel based on the ordered rooted tree data structure. Finally, we define a recursive transformation algorithm from a liveness formula to a statechart. This paper not only provides these theoretical results but also an implementation using the Human-Usable Textual Notation (HUTN) specification of OMG [11] and the Eclipse popular Integrated Development Environment (IDE).

This paper is organized in the following way. In section 2 we present the definition of the Gaia liveness formula followed by the formal definition of the statechart and its metamodel in section 3. The transformation algorithm and the technologies needed for implementing it are presented and discussed in section 4. Finally, section 5 includes conclusions and future work.

2   The Gaia Liveness Formula Definition

The Gaia methodology [19] is an attempt to define a general methodology that it is specifically tailored to the analysis and design of Multi-Agent Systems (MAS). Gaia emphasizes the need for new abstractions in order to model agent-based systems and supports both the levels of the individual agent structure and the agent society in the MAS development process. MAS, according to Gaia, are viewed as being composed of a number of autonomous interactive agents that live in an organized society in which each agent plays one or more specific roles. Gaia defines the structure of MAS in terms of the role model. The model identifies the roles that agents have to play within the MAS and the interaction protocols between the different roles. The Gaia methodology is a three phase process and at each phase the modeling of the MAS is further refined. These phases are the analysis phase, the architectural design phase and, finally, the detailed design phase.

The objective of the Gaia analysis phase is the identification of the roles and the modeling of interactions between the roles found. Roles consist of four attributes: *responsibilities*, *permissions*, *activities* and *protocols*. Responsibilities are the key attribute related to a role since they determine the functionality. Responsibilities are of two types: *liveness properties* – the role has to add something good to the system, and *safety properties* – the role must prevent something bad from happening to the system. Liveness describes the tasks that an agent must fulfill given certain environmental conditions and safety ensures that an acceptable state of affairs is maintained during the execution cycle. In order to realize responsibilities, a role has a set of permissions. Permissions represent what the role is allowed to do and, in particular, which information resources it is allowed to access. The activities are tasks that an agent performs without interacting with other agents. Finally, protocols are the specific patterns of interaction, e.g. a seller role can support different auction protocols. Gaia has operators and templates for representing roles and their attributes and also it has schemas that can be used for the representation of interactions between the various roles in a system. The operators that can be used for liveness expressions-

formulas along with their interpretations are presented in Table 1. Note that activities are written underlined in liveness formulas.

Table 1. Gaia Operators for Liveness Formulas

Operator	Interpretation	Operator	Interpretation
x‖y	x and y interleaved	x.y	x followed by y
x^ω	x occurs infinitely often	[x]	x is optional
x*	x occurs 0 or more times	x|y	x or y occurs
x+	x occurs 1 or more times		

The Gaia2JADE process [9] used the Gaia models and provided a roadmap for transforming Gaia liveness formulas to Finite State Machine (FSM) diagrams and then provided some code generation for JADE implementation. It also proposed some changes to Gaia such as the incorporation of a functionality table, where the activities were refined to algorithms, and a way to describe simple protocols. However, it did not cater for parallelism, and it did not produce the FSM diagrams automatically.

The reader can see a Gaia roles model for a role named "personal assistant" in Figure 1. This role employs seven activities and seven protocols (activities are underlined in the *Protocols and Activities* field). In its liveness formula it describes the order that these protocols and activities will be executed by this role.

The liveness formula grammar has not been defined formally in the literature, thus it is defined here using the Extended Backus–Naur Form (EBNF), which is a metasyntax notation used to express context-free grammars. It is a formal way to describe computer programming languages and other formal languages. The EBNF syntax for the *liveness* formula is presented in the following listing, using the BNF style followed by Russel and Norvig [16], i.e. terminal symbols are written in bold:

```
liveness               → { formula }
formula                → leftHandSide = expression
leftHandSide           → string
expression             → term
                        | parallelExpression
                        | orExpression
                        | sequentialExpression
parallelExpression     → term || term || … || term
orExpression           → term | term | … | term
sequentialExpression   → term . term . … . term
term                   → basicTerm
                        | (expression)
                        | [expression]
                        | term*
                        | term+
                        | termω
                        | |termω|number
```

```
basicTerm              → string
number                 → digit | digit number
digit                  → 1 | 2 | 3 | …
string                  → letter | letter string
letter                 → a | b | c | …
```

Role: Personal Assistant

Description: This role interacts with a meetings manager role in order to arrange and negotiate the user's meetings and with the user through a human-machine interface in order to get the user's requests and show him his schedule.

Protocols and Activities: <u>get user request</u>, <u>read schedule</u>, <u>show results</u>, <u>learn user preference</u>, <u>update user preferences</u>, send change request, receive change results, send new request, receive new results, receive proposed date, <u>decide response</u>, send results, receive outcome, <u>update schedule</u>

Responsibilities:

    Liveness:

        personal assistant = (manage meetings. learn user habits)$^\omega$ || (negotiate meeting date)$^\omega$

        manage meetings = get user request. (read schedule | request change meeting | request new meeting). show results

        learn user habits = learn user preference. update user preferences

        request change meeting = send change request. receive change results

        request new meeting = send new request. receive new results

        negotiate meeting date = receive proposed date. (decide response. send results. receive outcome)+. update schedule

---

Fig. 1. The Gaia role model of a personal assistant agent.

The reader should note that the Gaia operators have been enriched with a new operator, the $|x^\omega|^n$, with which we can define an activity that can be concurrently instantiated and executed more than one times (n times).

Figure 1 shows that the functionality of the personal assistant role is described by the liveness property. Thus, if the liveness formulas are transformed to a computer program then a large portion of the agent program is complete. However, this is not possible as there is a lot of information missing. First of all the functionality behind each activity is obscure. Then, the variables that will determine, e.g. whether the optional activities will be executed (i.e. an activity in brackets) are missing. This kind of information can be inserted in a statechart, thus we decided that in order to provide a design artifact that could lead to code generation we needed to transform the Gaia liveness formulas to a statechart [5]. However, before defining this transformation we needed a formal model for the statechart.

3   The Statechart Definition and Metamodel

Statecharts [5] are used for modeling systems. They are based on an activity-chart that is a hierarchical data-flow diagram, where the functional capabilities of the system are captured by activities and the data elements and signals that can flow between them. The behavioral aspects of these activities (what activity, when and under what conditions it will be active) are specified in statecharts. The fact that the statechart can capture together the functional and behavioral aspects of a system is its greatest advantage, as it completely defines a system. This is not true for a single UML model as a number of different models need to be combined for a complete description of a system (e.g. a class diagram together with an activity diagram). Thus, statecharts are ideal for defining systems in a platform independent manner. We intend to use statecharts in a specific level of abstraction, that of an agent, in order to model the interactions between its components (or capabilities). The statechart, therefore, implements the *intra-agent control model* (IAC) of an agent.

The authors in [5] present the statechart language adequately but not formally. Several authors have presented formal models for this language; as such an approach is needed for developing relevant statecharts-based Computer-Aided Software Engineering (CASE) tools. For example, David et al. [2] proposed a formal model for the RHAPSODY tool and Mikk et al. [8] for the STATEMATE tool. The first one has been used as basis for the definition of our statechart as it is the first intended for object-oriented language implementation (STATEMATE is for C language development). These models not only formally describe the elements of the statechart; they also focus on the execution semantics. However, this issue is out of the scope of this work. It is assumed that, as long as the language of statecharts is not altered, a statechart can be executed with any CASE tool.

The formal model that is adopted here-in is a subset of the ones presented in the literature as there are several features of the statecharts not used herein, such as the history states (which are also defined differently in these works). After formally presenting the statechart in the following paragraph, we will provide a metamodel in a common format such as the Eclipse Modeling Framework (EMF) and also discuss why this is needed.

3.1   Formal Statechart Definition

An *ordered rooted tree* is a rooted tree where the children of each internal vertex are ordered [15]. To produce a total order of the vertices of an ordered rooted tree all the vertices must be labeled. This is achieved recursively as follows:

1. Label the root with the integer 0. Then label its k children (at level 1) from left to right with 0.1, 0.2, 0.3, …, 0.k.
2. For each vertex v at level n with label A, label its k_v children, as they are drawn from left to right, with $A.1$, $A.2$, …, $A.k_v$.

Thus, A.1 means that A is the parent of A.1. The definition below for the statechart is inspired by the definition proposed by David et al. [2].

Definition 1. A *statechart* is a tuple (*L, δ*) where:

- *L* = (*S, λ, Var, Name, Activity*) is an ordered rooted tree structure representing the states of the statechart.

  — *S*⊆ℕ* is the set of all nodes in the tree.
  — *λ*: *S*→{AND, OR, BASIC, START, END, CONDITION} is a mapping from the set of nodes to labels giving the type of each node. For *l*∈*S* let *AND(l)* denote that *λ(l)*=AND. Similarly *OR(l)* denotes that *λ(l)*=OR and the same holds for all labels. START and END denote those nodes without activity, which exist so that execution can start and end inside OR-states. BASIC corresponds to a basic state. A condition state is denoted as CONDITION. START, END, BASIC and CONDITION nodes are leaves of *L*.
  — *Var* is a mapping from nodes to sets of variables. *var(l)* stands for the subset of local variables of a particular node *l*.
  — *Name* is a mapping from nodes to their names. *name(l)* stands for the name of a particular node *l*.
  — *Activity* is a mapping from nodes to their algorithms in text format implementing the processes of the respective states. *activity(l)* stands for the algorithm of a particular state that is represented by node *l*.

- *δ* ⊆ *S* × *TE* × *S* is the set of state transitions, where *TE* is a set of transition expressions.

  The following are also defined according to the definitions of David et al. (2003):

  Definition 2. Let *l* an internal vertex of an ordered rooted tree *L*. We call *sons(l)* = {*l.x* ∈ *S*|*x* ∈ ℕ} the children of *l*

  Definition 3. Let *l, k* two vertices of an ordered rooted tree *L* such that ∃*x*∈ℕ, *k.x* = *l*. Then the vertex *k* is called parent to *l* and it is denoted as *parent(l)*

  Definition 4. Let *l* a vertex of an ordered rooted tree *L*. Then, the ancestors of *l* are defined as *ancestors(l) = parent(l)* ∪ *ancestors(parent(l))*

3.2   The Statechart Metamodel

Model driven engineering relies heavily in model transformation [17]. Model transformation is the process of transforming a model to another model. The requirements for achieving the transformation are the existence of metamodels of the models in question and a transformation language in which to write the rules for transforming the elements of one metamodel to those of another metamodel.

In the software engineering domain a *model* is an abstraction of a software system (or part of it) and a *metamodel* is another abstraction, defining the properties of the model itself. Thus, like a computer program conforms to the grammar of the

programming language in which it is written, a model conforms to its metamodel (or its *reference model*). However, even a metamodel is itself a model. In the context of model engineering there is yet another level of abstraction, the *metametamodel*, which is defined as a model that conforms to itself [6].

A transformation that is used for transforming a textual representation to a graphical model is called a *Text to Model (T2M)* transformation. The textual representation must adhere to a language syntax definition usually using BNF. A liveness formula proposes such a kind of syntax. The graphical model must have a metamodel. Then, a transformation of the text to a graphical model can be defined.

In the heart of the model transformation procedure is the Eclipse Modeling Framework (EMF, [1]). EMF unifies Java, XML, and UML technologies, allowing the modeler to switch between them as they provide the same information in a different representation. Regardless of which one is used to define it, an EMF model is the common high-level representation that "glues" them all together.

Ecore [1] is EMF's model of a model (metamodel). It functions as a metametamodel and it is used for constructing metamodels. It defines that a model is composed of instances of the *EClass* type, which can have attributes (instances of the *EAttribute* type) or reference other EClass instances (through the *EReference* type). Finally, EAttributes can be of various *EDataType* instances (such are integers, strings, real numbers, etc). Figure 2 shows the ecore metamodel in detail.
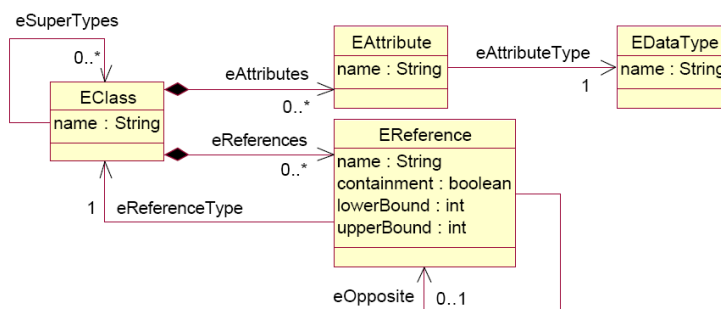

Fig. 2. The Ecore metamodel (Budinsky et al., 2003).

A similar technology, the Meta-Object Facility (MOF), is an OMG standard [10] for representing metamodels and manipulating them. There are a number of essential concepts used in MOF modeling. A Package is used to encapsulate a collection of related Classes and Associations. Packages can also contain simple type definitions. Classes exist in the commonly-used sense of the word, describing an object and its properties. These properties are represented through Attributes and References, which can be inherited using a multiple-inheritance system. Attributes have a name and a type. This includes a range of types from basic types such as integers, strings, and booleans to more complex types such as enumerations, and through to structured types. In addition, attributes have both upper and lower limits on the number of times that they can appear within a class instance. An Association is used to represent a

relationship between instances of two classes, each of which plays a role within the association. Associations can have the additional property of containment; an association represents a containment relationship if one of the participant classes does not exist outside the scope of the other. A Class participating in an association can also contain a Reference to the association. A reference appears much like an attribute, but reflects the set of class instances that participate in the Association with the containing class instance.

MOF is older than EMF and it influenced its design. MOF was initially designed primarily for use with the Common Object Request Broker Architecture (CORBA). CORBA is an architecture that enables programs, called objects, to communicate with one another regardless of what programming language they were written in or what operating system they're running on.

EMF, on the other hand, is a product of the Eclipse project, an open source project and was intended as a low-cost tool to obtain the benefits of formal modeling and Java code generation. As a consequence, one could say that EMF took a bottom-up approach whereas MOF took a top-down approach [4].

However, the EMF meta-model is simpler than the MOF meta-model in terms of its concepts, properties and containment structure, thus, the mapping of EMF's concepts into MOF's concepts is relatively straightforward and is mostly 1-to-1 translations. EMF is used today by a large open source community becoming a de facto standard in MDE. Moreover, third parties define MDE tools based on EMF technology, like the openArchitectureWare (oAW) platform for model-driven software development. For all these reasons it was decided that the EMF technology would be used.

The statechart metamodel (see Figure 3) contains nodes and transitions according to Definition 1. The metamodel defines a *Model* concept that has *nodes*, *transitions* and *variables* EReferences. Note that it also has a *name* EAttribute. The latter is used to define the namespace of the statechart. The namespace should follow the Java or C# modern package namespace format (see a sample namespace for the meetings management system in the next section with the transformations).

The nodes contain the following attributes (followed by the relevant concept name in the statechart definition):

- *name* (Name). The name of the node,
- *type* (λ). The type of the node (one of AND, OR, BASIC, START, END),
- *label* (label). The node's label, and
- *activity* (Activity). The activity related to the node.

Nodes also refer to *variables*. The Variable EClass has the attributes *name* and *type* (e.g. the variable with *name* "count" has *type* "integer"). Finally the transitions have four attributes:

- *name*, usually in the form <source node label>TO<target node label>
- *TE*, the transition expression
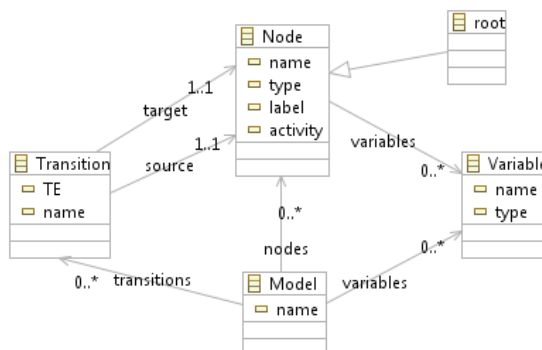- *source*, the source node label, and,
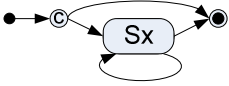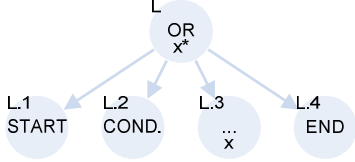- *target*, the target node label.
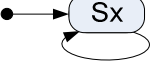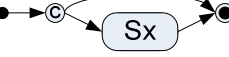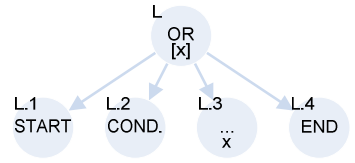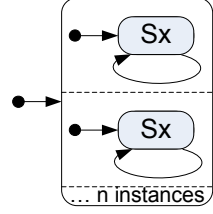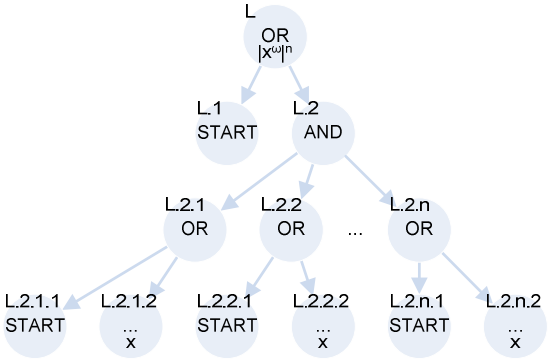
Fig. 3. The statechart metamodel.
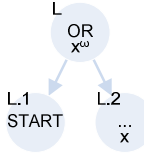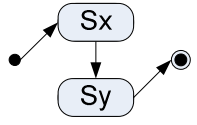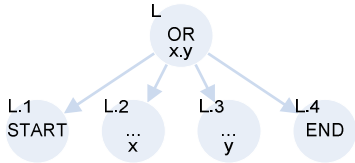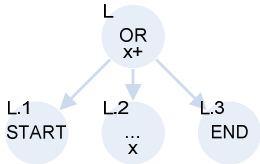
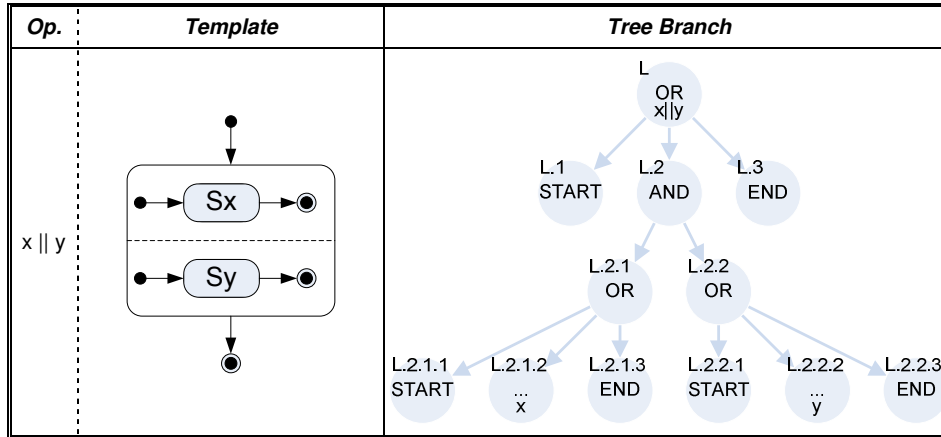## 4 The Liveness2Statechart Transformation

The Liveness2Statechart transformation is achieved by using the "Gaia operators transformation templates" (shown in Table 2) for transforming the process part of the agent interaction protocol model to a statechart. Table 2 has three columns. The first depicts a Gaia formula with a certain operator. The second shows how to draw the statechart relevant to this operator using the common statechart graphic language. The third shows how the same Gaia formula is transformed to the statechart representation defined in this paper (as a tree branch).

The tree branch representation (in Table 2) uses grey arrows to connect a father node to its sons. On the top left of each node the label of the node is shown. The root node of each branch is supposed to have a label *L* and the other nodes are labeled accordingly. The type of each node is written centered in the middle of the node. Finally, the name of each node is centered in the bottom of each node. The reader should note that the nodes for the *x* or *y* variables of the Gaia formula do not have a node type. This is because it is possible that they are basic or non-basic nodes. If they are basic then the node's type is set to BASIC, otherwise another branch is added with this node as the root and as the reader can notice all templates set the type of the root of the branch.

Table 2. Templates of extended Gaia operators (Op.) for Statechart generation

Op.	Template	Tree Branch
x^*		
x^ω		
$x . y$		
$x+$		
$[x]$		
$|x^\omega|^n$		

Op.	Template	Tree Branch
x ‖ y		

A designer can use the Gaia transformation templates to manually transform the liveness formula to a statechart. Alternatively, he can use an implementation of the following recursive algorithm for building the statechart automatically (three dots represent omitted code for space reasons):

```
Program transform(liveness)
  Var root = 0
  S = S ∪ {root}
  Name(root) = liveness->formula₁->leftHandSide
  createStatechart(formula₁->expression, root)
End Program

Procedure createStatechart(expression, father)
  Var terms = 0
  For each termᵢ in expression
    terms = terms + 1
  End For
  If terms > 1 Then
    If expression is sequentialExpression Then
      λ(father) = OR
      S = S ∪ {father.1}
      λ(father.1) = START
      Var k=2
      For Each termᵢ in expression
        S= S ∪ {father.k}
        Name(father.k) = termᵢ
        δ = δ ∪ {(father.(k-1), {}, father.k)}
        k = k + 1
      End For
      S = S ∪ {father.k}
      δ = δ ∪ {(father.(k-1), {}, father.k)}
```

```
      λ(father.k) = END
    Else If expression is orExpression
      ...
    Else If expression is parallelExpression
      ...
    End If
  For Each term_i in expression
    If term_i is basicTerm Then
      handleBasicTerm(term_i, getNode(father, term_i)
    Else
      If term_i is of type '('term')' Then
        createStatechart(term, getNode(father, term_i))
      Else If (term_i is of type '['term']') or (term_i is
of type term'*') Then
        ...
      Else If (term_i is of type term'^ω') or (term_i is of
type term'+') Then
        ...
      Else If term_i is of type '|'term'^ω|^n' Then
        ...
      End If
    End If
  End For
End function


Function getNode(father, term)
  QueuedList queue
  queue.addLast(father)
  Do While queue.notEmpty()
    element_i = queue.getFirst()
    If Name(element_i) = term Then Return element_i Else
      For each son_j in sons(element_i)
        queue.addLast(son_j)
      End For
    End If
  End do
End function


Function handleBasicTerm(term, node)
  Var isBasic = true
  For each formula_i in liveness
    If (formula_i->leftHandSide = term) Then
      createStatechart(formula_i->expression, node)
      isBasic = false
    End If
```

```
   End For
   If isBasic Then λ(node) = BASIC
End function
```

The program "transform" sets the root label equal to zero and its name equal to the left hand side of the first liveness formula. Then it calls the "createStatechart" procedure that takes two arguments. An *expression*, as it is defined in the Gaia liveness grammar, and a node (its label) under which it will build the tree.

The "createStatechart" procedure firstly checks whether the *expression* is a *parallelExpression*, an *orExpression* or a *sequentialExpression* and adds the relevant tree branch. Then, the procedure examines each *term* in the *expression*. A special function, the "handleBasicTerm" searches the formulas to find whether the term is a *basicTerm* or it appears in the left hand side of a following formula, which in this case needs to be expanded with the relevant tree branch. This is done by calling again the "createStatechart" procedure (recursively). Another function is used for this purpose, the "getNode". It searches (breadth first search) the tree branch below a node (the father) for the descendant with a specific name and returns its label. This is needed because the term's name is available but in order to add a tree branch the node's label is needed as a parameter for the "createStatechart" procedure call. If the examined *term* of the *expression* is a non-basic term then again the relevant tree branch is added to the statechart.

After applying the transformation algorithm, the statechart (or intra-agent control model) depicted in Figure 4 is created for the personal assistant liveness property presented in Figure 1. The reader can see the "negotiate meeting date" OR state (representing the execution of an interaction protocol) executed in parallel with the other agent capabilities.

For automating the transformation procedure we needed to implement this algorithm and produce statecharts adhering to the statechart metamodel. This is a T2M transformation. In order to do this we used a Java program for transforming the liveness property to a standardized textual representation. The latter could be automatically transformed to a statechart model based on Eclipse and EMF technology as it is described below.

Rose et al. [14] described an implementation of the Human-Usable Textual Notation (HUTN) specification of OMG [11] using Epsilon, a suite of tools for MDE for Eclipse. OMG created HUTN aiming to offer three main benefits to MDE:

— a generic specification that can provide a concrete HUTN language for any model, which is described by a metamodel
— the HUTN languages to be fully automated both for production and parsing
— the HUTN languages to conform to human-usability criteria

The Epsilon platform is an implementation of HUTN, which automates the transformation process by eliminating the need for a grammar specification by auto defining it accepting as input the relevant EMF metamodel (i.e. the one shown in Figure 3). This is the main reason for choosing HUTN. In Figure 5, the eclipse project for the realization of the Liveness2Statechart transformation is presented. It is a simple Java project where the HUTN nature has been turned on (by right-clicking on

the project icon on the Package explorer). The input for this transformation is the Gaia roles model liveness property in text format, adhering to the grammar presented in §2.
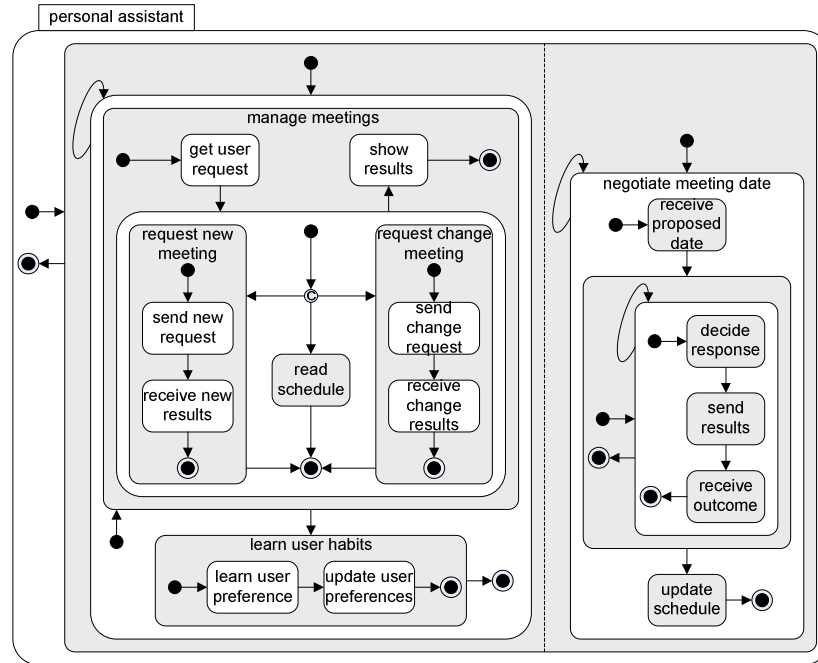

Fig. 4. The automatically generated statechart for the personal assistant agent.

The presented transformation algorithm has been implemented in the java language. It transforms the liveness formula of an SRM role to a HUTN file. The usage of the HUTN technology also helped a lot in debugging the algorithm as the output was in human-readable format. The modeler just has to execute the "Liveness2HUTN.java" file in order to create the HUTN representation of the statechart model (shown in Figure 5). Then, simply by right-clicking to the HUTN file the modeler can generate the statechart model. An extract of this model where the XML elements representing the HUTN representation part visible in Figure 5 is the:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.
org/XMI" xmlns:IAC="http://mi.parisdescartes.fr/ASEME/
metamodels/IAC">
  <IAC:Node name="open_group_ReadSchedule_or_Request
  ChangeMeeting_or_RequestNewMeeting_close_group" type=
  "OR" label="0.2.1.2.2.2.3" activity="null" />
  <IAC:Node name="GetUserRequest" type="BASIC"
  label="0.2.1.2.2.2.2" activity="null" />
...
```

Thus, the statechart model has now been initialized with the information available in the Gaia roles model and it can be refined in the design phase using, e.g., the *Sample Reflective Ecore Model Editor* of Eclipse.
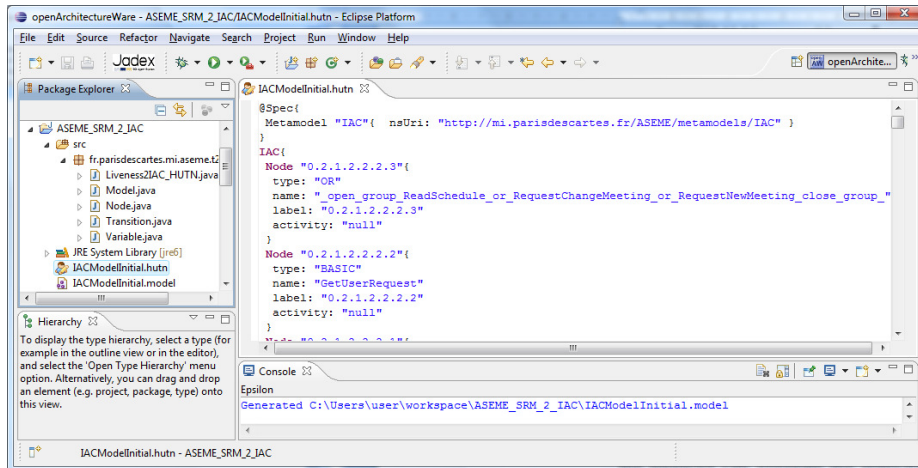

Fig. 5. The Eclipse project for T2M transformation.

5 Conclusion

This paper showed how engineers, who use the Gaia methodology for modeling their agent-based systems, can implement their agents through the use of statecharts. The later allow to define the interactions between the different modules (or capabilities) of an agent (i.e. his intra-agent control) in a sufficient detail that can lead to implementation. A statechart is a platform independent model (PIM) of the system under development, as statecharts can be implemented using a number of existing programming languages and CASE tools. The statechart is automatically produced by the Gaia liveness property (a set of liveness formulas), which describes the behavior of an agent. This transformation is not a straightforward process and it is achieved through the following original results:

- Definition of a grammar for representing a liveness model.
- Formal definition of a statechart for agent-oriented development.
- Conception of a recursive algorithm for transforming the Gaia liveness property to a statechart. The modeler can make the transformation either manually (using the Gaia transformation templates) or automatically using the popular Eclipse IDE.

The manual transformation is also a valuable result as a developer can transform the liveness property to a statechart using any existing CASE tool. The Rhapsody tool [5] has been successfully used for implementing the MARKET-MINER agent, a real world system [18]. We are currently working in automating the code generation process (model to text – M2T) for a popular agent platform, the Java Agent Development Framework (JADE).

References

1. Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Brodsky, S.A., Grose, T.J.: Eclipse Modeling Framework. Addison Wesley (2003)
2. David, A., Deneux, J., d'Orso, J. : A Formal Semantics for UML Statecharts. Technical Report 2003-010, Uppsala University (2003)
3. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández, R.: Model Transformations for Improving Multi-agent Systems Development in INGENIAS. In: 10th International Workshop on Agent-Oriented Software Engineering (AOSE'09), Budapest Hungary (2009)
4. Gerber, A., Raymond, K.: MOF to EMF: there and back again. In: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange (eclipse '03), pp. 60--64. ACM Press, New York (2003)
5. Harel, D., Kugler, H.: The RHAPSODY Semantics of Statecharts (Or on the Executable Core of the UML). In: Integration of Software Specification Techniques for Applications in Engineering. LNCS, vol. 3147, pp. 325—354. Springer, Heidelberg (2004)
6. Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006). LNCS, vol. 4037, pp. 171--185. Springer, Heidelberg (2006)
7. Kleppe, A., Warmer, S., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
8. Mikk, E., Lakhnech, Y., Petersohn, C., Siegel, M.: On formal semantics of Statecharts as supported by STATEMATE. In: Proceedings of the second BCS-FACS Northern Formal Methods Workshop. Springer-Verlag (1997)
9. Moraitis, P., Spanoudakis, N.: The Gaia2JADE Process for Multi-Agent Systems Development. J. Appl. Artif. Intell. 20(2-4), 251--273 (2006)
10. Object Management Group: Meta Object Facility (MOF) Core Specification (2001)
11. Object Management Group: Human-Usable Textual Notation V1.0 (2004)
12. openArchitectureWare (oAW), http://www.openarchitectureware.org/
13. Perini, A., Susi, A.: Automating Model Transformations in Agent-Oriented Modeling. In: Agent-Oriented Software Engineering VI. LNCS, vol. 3950, pp. 167--178. Springer (2006)
14. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: Constructing models with the Human-Usable Textual Notation. In: 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS). LNCS, vol. 5301, pp. 249--263. Springer (2008)
15. Rosen H.K. Discreet Mathematics and its Applications. Forth edition, McGraw Hill (1999)
16. Russel, S., Norvig, P.: Artificial Intelligence a Modern Approach. Second Edition, Prentice Hall (2003)
17. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Softw. 20(5), 42--45 (2003)
18. Spanoudakis N., Moraitis, P.: Automated Product Pricing Using Argumentation. In: Proceedings of the 5th IFIP Conference on Artificial Intelligence Applications & Innovations (AIAI 2009). Springer (2009)
19. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: the Gaia Methodology. ACM T. Softw. Eng. Meth. 12(3), 317--370 (2003)