Formal Development of Multi-Agent Systems with FPASSI: Towards Formalizing PASSI Methodology using Rewriting Logic

Mihoub Mazouz
Department of Mathematics and Computer Science, RELA(CS)^2 Laboratory
University of Larbi Ben M'Hidi, Oum El Bouaghi, Algeria
E-mail: mazouz_mihoub@hotmail.fr

Farid Mokhati
Department of Mathematics and Computer Science, RELA(CS)^2 Laboratory
University of Larbi Ben M'Hidi, Oum El Bouaghi, Algeria
E-mail: mokhati@yahoo.fr

Mourad Badri
Department of Mathematics and Computer Science, Glog Laboratory
University of Quebec, Trois-Rivières, Canada
E-mail: Mourad.Badri@uqtr.ca

Keywords: formal development of MAS, PASSI, validation, verification, rewriting logic, maude, maude-strategy, model-to-text transformation

Received: June 20, 2016

Agent technology has proved its ability and efficiency in modelling complex distributed applications. During the last two decades, several MAS development methodologies have been proposed like, for instance, Gaia, Tropos and PASSI. Although these methodologies have made significant contributions to meet several challenges in the MAS development field, most of them do not use formal techniques. Formal methods, as it is well known, play a significant role in developing more reliable and robust MAS. This paper presents the Formal-PASSI methodology. Formal-PASSI is an extension of the well-known PASSI methodology. The extension consists mainly of the integration of a new formal model to the design process. The new model is based on the Maude language and its extension Maude-Strategy. It aims at offering a formal description of the MAS under development by a Model-to-Text transformation. The generated formal description is then used to validate some PASSI behavioural diagrams and check properties of both single & multi-agent abstraction levels before passing to the code model. The integration of formal methods into PASSI design process seems a good way to ensure the development of high quality agent-based applications. The proposed approach is supported by a tool (F-PTK) that we have developed and illustrated throughout the ATM case study.

Povzetek: V članku je predstavljena formalna PASSI MAS metodologija, tj. multi-agentna metodologija.

1 Introduction

Current computing systems became increasingly complex with high safety requirements. Agent technology has proved its ability and efficiency in modelling complex distributed applications. As well as any other technology, the emergence of the agent technology pushes the research community to propose new methodologies, languages and tools to support it and to enable a wider spread in the industry sector. Many methodologies like PASSI [1,2], Gaia [3,4], ADELFE [5,6,7], Prometheus [8], Tropos [9] and INGENIAS [10] have been proposed to facilitate and to assist the development of Multi-Agent Systems (MAS). Although these methodologies have made real progress in the MAS development field, proposing new methodologies that assist agent-based systems development is still insufficient for industrial adoption [11].

The development of such systems requires solid bases in terms of specification. Existing methodologies use abstract and/or semi-formal specifications. Although such types of specifications offer several advantages such as the readability and the facility of comprehension, they have drawbacks like ambiguity and inconsistency, which are manually difficult to detect. However, formal specifications face these drawbacks and enable the description of the system under development in a precise and unambiguous way. Using formal methods is essential to produce high quality agent-based systems at the end of the development process. In particular, integrating formal methods into the development process of MAS methodologies leads to the production of reliable systems.

In order to overcome the problems quoted above, many proposals are trying to use formal methods in agent-oriented software engineering (AOSE) (see Section 2). However, most of them present several limitations, especially; they do not use formal methods within an entire design process. Moreover, many of them are not supported by adequate tools.

PASSI (Process for Agent Societies Specification and Implementation) [1, 2] is a step-by-step requirement-to-code methodology for designing and developing agent-oriented systems that integrates concepts from both Object-Oriented Software Engineering (OOSE) and MAS using UML (Unified Modelling Language) notation. PASSI covers almost of development process stages, and can be used to assist the development of general-purpose agent-oriented systems although it has evolved from a long period experiment to the development of embedded robotics applications [12]. However, being PASSI based on a semi-formal language such as UML makes the validation and verification activities less efficient.

In this paper, we propose F-PASSI (Formal-PASSI), a formalization of the PASSI methodology by adding a new formal model into its design process. The extension is based on rewriting logic [13, 14] and particularly the Maude language [15,16] (and its extension Maude-Strategy [17]). The integrated model aims at offering a Maude-based formal description of the MAS under development to enrich the semantic of its UML-based design. The produced formal description is then exploited to validate PASSI behavioural diagrams (some of them until now) by formal simulation thanks to Maude, and Maude LTL model-checker [18] in order to verify system properties in both single/multi agent abstract levels. A tool was developed to support our approach.

The remainder of this paper is organized as follows: In section 2, we give an overview of major related works. In section 3, we give a brief description of rewriting logic as well as Maude language (and its extension Maude-Strategy). In section 4, a brief description of the PASSI methodology is given. We introduce, in section 5, the proposed formal extension for PASSI. Our developed tool is shown in section 6. In section 7, the ATM case study is used to illustrate our approach. Finally, section 8 gives some conclusions and future work directions.

2 Related works

Using formal methods in multi-agent systems development is a challenge raised by many researchers in MAS area. El Fallah-Seghrouchni et al. have presented a classification of the proposed works on formal development of MAS [19]. According to the authors, three alternatives can be captured from the literature: (A) Formal derivation: which is a kind of model-to-code transformation and aims at realizing MAS based on a given specification. (B) Enhancement of an existing methodology by integrating formal meanings to its design. (C) Proposing a new one. The fact that our work can be considered as an integration of formal methods to an existing methodology, PASSI, makes our focus in this section on works belonging to the second category.

In [20, 21], Ball et al. have presented an incremental development process using Event-B [22] for multi-agent systems. The proposed process can be divided into two stages. In the first one, informal models based on agent concepts are constructed. In the second stage, based on the informal models, the Event-B models are constructed by the developer, which is provided by guidance to make the transformation from informal design to formal models straightforward. The constructed Event-B models are refined and decomposed into specifications of roles. In [23], a set of modelling patterns providing fault-tolerance in Event-B models of multi-agent interactions are presented. Another work proposing a new formal methodology is ForMAAD [24, 25]. ForMAAD is a model driven approach for designing agent-based application. It uses Agent Modelling Language (AML) [26] to model architectural and behavioural concepts associated with multi-agent systems; and Temporal Z [27] to guarantee a formal verification of the models. Extensions of StartUML[1] tool are made to support the models they proposed.

Two works using formal methods for the Tropos methodology [9] can be emphasized here. First, Fuxman et al. [28] have proposed an extension of Tropos, Formal Tropos, with a formal specification of early requirements. For that, Formal Tropos language is defined by integrating the primitive concepts of Tropos with a temporal specification language inspired by KAOS [29]. After the translation (using the implemented T-tool[2]) of the requirements specification written by the analyst into an intermediate language, an enhanced version of NuSMV model checker [30] performs consistency checking ("the specification admits valid scenarios"), possibility checking ("there is some scenarios for the system that respect certain possibility properties") and assertion validation ("all scenarios for the system respect certain assertion properties"). Secondly, in [31], a mapping of β-Tropos concepts [32] into the computational logic-based framework SCIFF [33] is defined and important formal properties (soundness, completeness and termination) are identified and discussed. The formal specifications are verified using SCIFF engine. Instead of writing it manually, as in the last works, the formal specification is produced in a systematic way in Formal-PASSI thanks to F-PTK (Formal-PASSI Tool Kit), the tool we have developed, this makes it, unlike Formal Tropos, less based on the subjective judgment of the developer. Also, in Formal-PASSI, the formal specification combines, in addition to the domain knowledge, the structure and behaviour of agents composing the MAS to be exploited later to validate and verify its correctness.

Instead of proposing new formal methodologies for MAS development or enhancing existing ones, other researchers have used formal methods, separately from any methodology, for particular design aspects. Fadil et al. [34] have used the B method [35, 36] to formally model interactions between agents in order to check and then prove the initial UML specification. The approach was

[1] http://staruml.io

[2] http://disi.unitn.it/~ft/ft_tool.html

illustrated using Contact-Net protocol as a case study. Jemni Ben Ayed et al. [37] have presented a specification and verification technique for interaction protocols in MAS by combining AUML (Agent UML) [38] and Event-B method [22]. In their technique, the interaction protocol is modelled in an AUML protocol diagram and translated in Event B. The required IPs (Interaction Protocols) safety and liveliness properties are added to the derived specification for verification using the B4free tool[1].

As B method, the Z language [39] and its extension Temporal Z [27] have been the subject of many works. In [40], the authors have presented a formal approach using Temporal Z in two phases. In the specification phase, user requirements are described in an abstract way avoiding the description of implementation details. Then, based on a succession of refinements, the design phase aims at inventing a set of inter-agent (collective) behaviours as well as intra-agent (individual) behaviours, which have to satisfy the user requirements. Other works address the use of formal methods in runtime to verify some properties (that are not verifiable in design phase) as in [41], where a JADE-based formal verification methodology for MAS in semi-runtime approach has been proposed. The proposed verification process used timed trace theory to detect time constraint failures.

Lapouchnian et al. [42] have proposed a combined agent-oriented requirements engineering approach using informal i* [43] models, ConGolog [44] and (its extension) CASL [45] formal specifications. Social dependencies between agents are modelled using the i* framework. This framework is used to perform an analysis of opportunities and vulnerabilities. The models are gradually made more precise by using annotated models (Annotations are introduced in [46] and extended in [47]). After that, complex processes can be formally modelled using ConGolog or CASL with subsequent verification or simulation.

In [48], the authors have presented an extension of G-net formalism [49] (a type of high level Petri net) called Agent-oriented G-net to serve as high level design of intelligent agents by means of their internal states, their environment, their interactions, etc. Based on this high level design, agent architecture and detailed design for agent implementation can be derived using the ADK tool they developed. Stamatopoulou et al. [50] have presented an open framework facilitating formal modelling of multi-agent systems called OPERAS by employing two existing formal methods: X-machines [51] and PPS (Population P Systems) [52]. By using this framework, agent's behaviour can be formally modelled and controlled over its internal states, as well as the mutations that occur in the structure of a MAS. The authors have applied the framework to swarm systems.

Compared to the works discussed above, the approach we propose: (1) integrates formal methods, not separately from any methodology, but into an entire design process (PASSI design process), (2) is based on a powerful formal language, Maude, which offers many tools as Maude LTL model checker [18], (3) checks the specified properties before passing to code details, (4) is supported by a tool (F-PTK) which offers many services such as automating the production of the Maude-based formal description of the MAS under development by means of its structure (Agents, roles, tasks, action tasks) and the domain knowledge.

3 Rewriting Logic, Maude & Maude-Strategy

3.1 Rewriting Logic

The rewriting logic was introduced by Jose Meseguer [13, 14] to describe concurrent systems. It makes it possible to think in a correct manner on the concurrent systems having states and evolving in terms of transitions. Indeed, the rewriting logic unifies several formal models which express concurrency as labelled transition systems [53], Petri nets [54] and CCS [55]. The basic statements of this logic are called rewriting rules and have the form: t ⟶ t' if C, where t and t' are algebraic terms describing a partial state of the concurrent system. A rewriting rule, in this case, describes a change of a partial state towards another if a certain condition C is true. Formally, a theory of rewriting is a triplet R = (∑,E,R) where:

- (∑, E) an equational theory with function symbols ∑ and equations E;
- R a set of labelled rewrite rules. These rules are of the form: $t \to t'$ (unconditional rewriting rules) or $t \to t'\ if\ condition$ (conditional rewriting rules).

The unconditional rewriting rules indicate that: the term t becomes t', but, the conditional rewriting rules indicate that: t becomes t' if a certain condition is true. A theory of rewriting has a set of inference rules [13, 14]:

- Reflexivity: For each [t] ∈ T∑, E (X), $\frac{}{[t]\to[t']}$
- Congruency: For each $f \in \sum n, n \in N$

$$\frac{[t_1] \to [t'_1] \dots [t_n] \to [t'_n]}{[f(t_1, \dots, t_n)] \to [f(t'_1, \dots, t'_n)]}$$

- Replacement: For each rewriting rule:

r: [t(x1,…, xn)] → [t'(x1,…, xn)] in R,

$$\frac{[w_1] \to [w'_1] \dots [w_n] \to [w'_n]}{[t(\overline{w}/x)] \to [t'(\overline{w}'/x)]}$$

Such as $t(\overline{w}/x)$ indicates the simultaneous substitution of w_i for x_i in t.

- Transitivity: $\frac{[t_1]\to[t_2] \quad [t_2]\to[t_3]}{[t_1]\to[t_3]}$.

Figure 1 visualizes each one of these rules.

[1] http://www.b4free.com

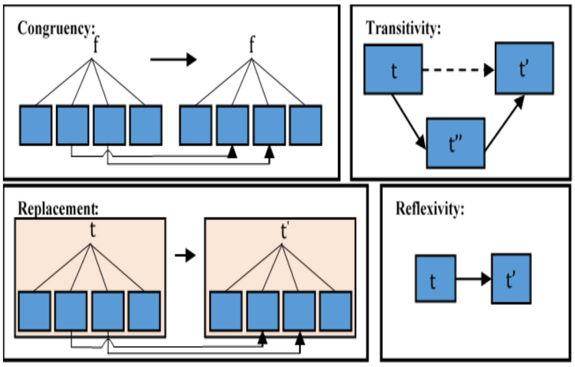


Figure 1: Visualization of inference rules of a rewriting theory [14].

Among the languages implementing the rewriting logic, we quote *CafeOBJ*[1] **[56]** and Maude **[15, 16]**.

3.2 Maude language

Defined by J. Meseguer, the Maude language **[15, 16]** is one of the most powerful implementations of the rewriting logic. Maude is a high level, very powerful, declarative language for the construction of the various kinds of applications based on both equational and rewriting logics. It offers few syntactic constructions and well-defined semantics. The basic unit of specification and programming in Maude is the module. In fact, there are three types of modules:

Functional modules: Define the sorts of data and the operations on these data through equational theories. The sorts of data are composed of elements which can be called by terms. A functional module is declared according to the following syntax:

```
fmod MODULE-NAME is
     …
endfm
```

System modules: Specify a rewriting theory. A system module has sorts, operations and can have equations and rewriting rules, which can be conditional. A System module is declared as follows:

```
mod MODULE-NAME is
     …
endm
```

The addition that a system module offers (compared to a functional module), is the ability of specifying rewriting rules. The unconditional rules are declared as follows:

rl [<Label>] : <Term1> => <Term2> .

The conditional rewriting rules can have very general conditions implying equations and other rewritings. In their representation in Maude, the conditional rules are declared as follows:

crl [<Label>] : <Term1> => <Term2>
if <Condition-1> and .. and <Condition-k> .

Object-oriented modules: Compared to system modules, object-oriented modules offer a more suitable syntax to describe the basic entities of the object paradigm as, among others: classes, objects, messages and configurations. An object-oriented module is declared according to the following syntax:

```
omod MODULE-NAME is
     …
endom
```

Figures 2, 3 and 4 show an example for each Maude's module type (independent modules).

```
fmod COORD-COMPLEX-TYPE is
 inc FLOAT .
 inc BOOL .
 sort Coord .
 op _;_ : Float Float -> Coord .
 op empty : -> Coord .
 op getLatitude : Coord -> Float .
 op getLongitude : Coord -> Float .
 op equals : Coord Coord -> Bool .
 vars Lat Lon x1 y1 x2 y2 : Float .
 eq getLatitude ( Lat ; Lon ) =  Lat .
 eq getLongitude ( Lat ; Lon ) = Lon .
 eq equals ( x1 ; y1  , x2 ; y2 ) = if (x1 == x2) and
(y1 == y2) then true else false fi .
endfm
```

Figure 2: Example of a functional module.

```
omod ACCOUNT-CONCEPT is
  pr STRING .

  class Account | accountN : String, owner : Oid .
endom
```

Figure 3: Example of a system module [16].

```
mod BB-TEST is
 sort Expression .
 ops a b bingo : -> Expression .
 op f : Expression Expression ->
                         Expression .
 rl a => b .
 rl b => a .
 rl f(b, b) => bingo .
endm
```

Figure 4: Example of an Object-oriented module.

It is important to note that there exist two separated levels in the current version of Maude (Maude 2.7): Core Maude and Full Maude.

Core Maude: It is the basic level of Maude, programmed directly in C++. It implements all the basic functionalities of the language, the functional modules and the system modules;

Full Maude: Full Maude is the higher level. Programmed in Core Maude, it is actually used with object-oriented programming paradigm and using it offers the possibility of using object-oriented modules. All commands and modules in Full Maude must be declared between brackets.

[1] https://cafeobj.org/

Among Maude's characteristics that justify our choice of Maude, we quote:

- ✓ **Easy**: Programming with Maude is easy because it is a declarative language and it offers few and very simple syntactic constructions which are easy to be understood;
- ✓ **Having a strong semantics**: Based on a solid logic: rewriting logic;
- ✓ **Expressive**: Determinist and concurrent, non-determinist calculations can be expressed easily respectively by equations in functional modules and rewriting rules in system modules in Maude.
- ✓ **Wide spectrum**: It supports the formal specification, prototyping and concurrent programming.
- ✓ **Multi-Paradigm**: It combines functional, concurrent and object paradigms.
- ✓ **Executable**: A Maude specification is directly executable;
- ✓ **Equipped with many tools**: It offers to its users a set of tools[1] like: Declarative Debugger[2], the Anima tool[3] and The Maude LTL model checker [18].

Also, many extensions of Maude are developed as *Real-Time Maude* [57], and *Maude Strategy Language* [17].

3.3 Maude-Strategy

The Maude-Strategy [17] is an extension of Maude Language written in Maude itself. It was defined in order to explicitly control the way in which the rewriting rules are applied. The originality of Maude-Strategy language is to make it possible to specify the strategy of applying the defined rewriting rules, which makes it possible to clearly separate the transformation rules and their control. When we don't have such a language that can specify strategies controlling the order of applying the rewriting rules separately, the order of their application is often coded in the rewriting rules themselves, which makes more complex and less readable the program to be written. The treatment and control operations are mixed. The strategies are defined by using the modules of strategies.

It is possible to define many modules of strategies for only one system module (or object-oriented module) in order to express the various possible forms of rewritings. A Strategy E is described as an operation that, when it is applied to a given term *t*, produces consequently a set of terms (Eventually empty):

$$\_@\_ : Strat \times T_{\Sigma}(X) \rightarrow P(T_{\Sigma}(X))$$

This operation is extended to sets of terms so that:
if $T \subseteq P(T_{\Sigma}(X))$ *and* $E \in Strat$ *then* $E\ @\ T = \bigcup_{t \in T} S\ @\ t$.

For space reason, only a subset of strategies [17] is described:

Identity and Failure (Idle and fail): The first two basic strategies are the identity and the failure, defined by *Idle* and *fail*. The application of the identity strategy turns over the unchanged term:

$$Idle\ @\ t = \{t\}$$

The application of the strategy failure turns over the empty set as a result:

$$Fail\ @\ t = \emptyset$$

Elementary strategies: Starting from the labels of rules, it is thus possible to build strategies, which turn over one or more results, to schedule the application of the rules and to repeat as a long time as possible the application of a rule or a strategy. A labelled rule is thus regarded as an elementary strategy and the result of the application of a labelled rule *L* on a term *t* turns over the set reached terms by applying the rule *L*. If no rule labelled by *L* can be applied, it is said that the strategy failed.

Regular expressions: The expression of elementary strategies can be combined by using operators of concatenation (;), of union (|), of iteration (E* for zero or more iterations, E+ for one or more iteration).

```
op _;_ : Strat Strat -> Strat [assoc] .
op _|_ : Strat Strat -> Strat [assoc comm] .
op _* : Strat -> Strat .
op _+ : Strat -> Strat .
```

The application of the concatenation (;) of two strategies *E* and *E'* on a term *t* has as a result all the results of application of *E'* on the whole of all results of the application of *E* on *t*:

$$[(E\ ;\ E')\ @\ t] = [E'\ @\ [E\ @\ t]$$

On the other hand, the application of the union (|) of two strategies *E* and *E'* on a term *t* has as a result the whole of results of application of both *E* and *E'* separately on the term *t*:

$$[(E \mid E')\ @\ t] = [E\ @\ t]\ \cup [E'\ @\ t]$$

The operators of iteration (E^* and E^+) are used to define strategies, which concatenate successively the same strategy: $[E^+\ @\ t] = \bigcup_{i\geq 1}[E_i\ @\ t]$ *where* $E_1 = E$ *and* $E_n = (E\ ;\ E_{n-1})$ *for* $n > 1$.

$$[E^*@\ t] = [(idle \mid E+)\ @\ t].$$

The operators of iteration (E^* and E^+) are used to define strategies, which concatenate successively the same strategy: $[E^+\ @\ t] = \bigcup_{i\geq 1}[E_i\ @\ t]$ *where* $E_1 = E$ *and* $E_n = (E\ ;\ E_{n-1})$ *for* $n > 1$.

$$[E^*@\ t] = [(idle \mid E+)\ @\ t].$$

Conditional strategies: Moreover, Maude-Strategy defines operators of choice which take the following general form: if E then E' else E'' (where E ? E' : E")

This form when it is applied on a term t acts like the following: The strategy E is initially applied to t, if E is evaluated successfully ($E\ @\ t \neq \emptyset$), the strategy E' is applied on the set of terms which results from the evaluation of E, if not $E\ @\ t = \emptyset$, E" is applied to the initial term t. Among the derived operators from this general form, we distinguish the operator orelse which acts like the following: When E is applied successfully,

[1] http://maude.cs.illinois.edu/w/index.php?title=Maude_Tools

[2] http://maude.sip.ucm.es/debugging/

[3] http://safe-tools.dsic.upv.es/anima/

the result is obtained, but if that is failed, then E' is applied to the initial term. In other words: E orelse E' = if E then idle else E'.

Figure 5 illustrates a Strategy module. In this module, four strategies are identified: *Branch0, Branch1, Branch2 and Protocol*. For example, the strategy "*Branch1*" specifies that the rewriting rule labelled by "Send-The-Nearest-Ambulance" must be applied in parallel with the sequence of the two rules labelled respectively by "*Transformation-1*" and "*Send-Police-Patrol*".

```
(smod BUISNESS-PROCESS-PROTOCOL is

  strat Branch0 : @ Configuration .
  sd Branch0 := ( First-Order ; Is-Reported ) .
  strat Branch1 : @ Configuration .
  sd    Branch1 := ( Send-The-Nearest-Ambulance |
             ( Transformation-1 ; Send-Police-Patrol )  )! .
  strat Branch2 : @ Configuration .
  sd   Branch2 := ( Book-The-Nearest-Hospital ;
                    Branch1 ; Mark-Accident-As-Reported ) .
  strat Protocol : @ Configuration .
  sd    Protocol := ( Branch0 ; ( Transformation-2 orelse
                        Branch2 ) ) .
endsm)
```

Figure 5: Example of a strategy module.

4 PASSI Methodology

PASSI (Process for Agent Societies Specification and Implementation) [1, 2], is a step-by-step requirement-to-code methodology for designing and developing agent-oriented systems. It integrates concepts from both OOSE and artificial intelligence approaches using UML notation. It refers to the most diffused standards: *(A)UML, FIPA, JAVA, RDF* and it is composed of a complete incremental and iterative design process and modelling language that is an extension of UML. PASSI is based on reuse that is performed through design patterns and supported by the PTK tool (PASSI Tool Kit) [58]:

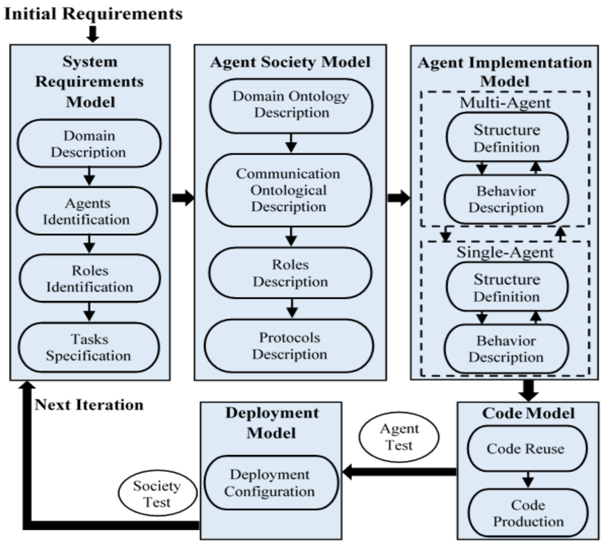


Figure 6: PASSI methodology [1, 2].

As Figure 6 shows, PASSI methodology is composed of five models and a test activity, each model contains one or more phases:

System Requirements Model: It is composed of four (4) phases. The functional requirements of the MAS are captured through a use case diagram (*Domain Description phase*). The agents carrying out these requirements are then identified *(Agents Identification phase)* via packaged use case diagram where each agent (package) is responsible of one or more requirement(s) (use case(s)). Roles played by agents in different scenarios are identified (*Roles Identification phase*) using sequence diagram where each life line signifies one played role by following the syntax: *<Role> : <AgentName>* and each scenario is explored by one sequence diagram. Finally, the 4th phase *(Tasks specification)* aims at specifying the different tasks of an agent and the relationships between them (internal tasks) and other agent's tasks (external tasks).

Agent Society Model: It is composed of four (4) phases. The knowledge (about the domain) of agents composing the system is described through an ontology (Concepts, predicates and actions) that is specified by a class diagram *(Domain Ontology Description phase)*. Communications between agents are described also by a class diagram (*Communication Ontological Description phase*) where agents are specified by classes and each communication between two agents is specified by an association class (with three attributes Ontology, Language, Protocol). The identified roles are described (*Roles Description phase*) by means of their own tasks, one class for each role, one operation for each task, and one package for each agent. Roles can be connected by relationships of type: *[ROLE_CHANGE], [SERVICE_DEPENDENCY] or [RESSOURCE or COMMUNICATION_AVAILABILITY]*. If the protocols used during communications are not standard, they will be specified via AUML sequence diagram (*Protocols Description phase*).

Agent Implementation Model: It is composed of two (2) phases. In this model, the structure of the system (*Multi/Single-agent Structure Definition phase*) is defined using a class diagram showing all agents composing the system by classes and theirs tasks by operations (for multi-agent point of view) and showing tasks by classes and their actions by operations (for single-agent point of view). The behaviour of the system (*Multi/Single-agent Behaviour Description phase*) is described by a specific activity diagram for multi-Agent point of view, and by state machine or other formalisms as flow charts for single-agent behaviour description **[1]**.

Code Model: It is composed of two (2) phases. In the first phase *(Code Reuse)*, design patterns already developed can be used directly. In the second phase (*Code Production)*, a skeleton of the system's source code is automatically generated by the PTK and a manual completion of the generated code is then achieved by the developer.

Deployment Model: It is composed of one phase *(Deployment Configuration)*. A deployment diagram is used to describe the allocation of agents to different processing units and any constraints on agent migration and mobility.

Test activity is divided into two different levels: 1) *single-agent test*: when a framework built on top of JADE is implemented [59]. The principal framework classes are: "Test" class for testing a specific task of an agent; "*TestGroup*" class for testing all tasks composing a specific agent. 2) *society test:* at this level, integration verification is carried out together with the validation of the overall results of the current iteration [1].

The meta-model adopted for PASSI MAS is divided into three areas [2]: *(1) Problem domain*: where the elements describing the requirements that will be achieved by the future system are included. These elements are directly connected to the System Requirements Model. *(2) Agency domain*: where the elements describing the multi-agent society in terms of environment (defined by a set of ontological elements) and the social aspect of agents (interaction between them) are included. The items of this area are connected directly to the Agent Society Model. *(3) Solution domain*: where the elements describing the architectural solution (respecting the architecture of FIPA) of the problem in terms of agent classes, task class, agent code and task code are included. The elements of this area are connected directly to the two models: Agent Implementation and Code.

5 Formal PASSI

The PASSI methodology is based on a semi-formal notation (UML). This makes the designed diagrams prone of containing incoherencies or inconsistencies and makes the testing activity less efficient. We have proposed Formal PASSI (see Figure 7), an extension of PASSI methodology, in order to formalize its diagrams, and to give the designer the ability to apply some formal techniques such as model-checking on the formal specification. As Figure 7 shows, a new model *(Formal Model*, in yellow color*)* is integrated in PASSI design process. The formal model is based on the rewriting logic and its Maude language (and its extension Maude-Strategy). It aims at offering a formal description of the MAS under development. This formal description is then exploited to apply formal validation and verification. Formal Model is composed of four (4) phases:

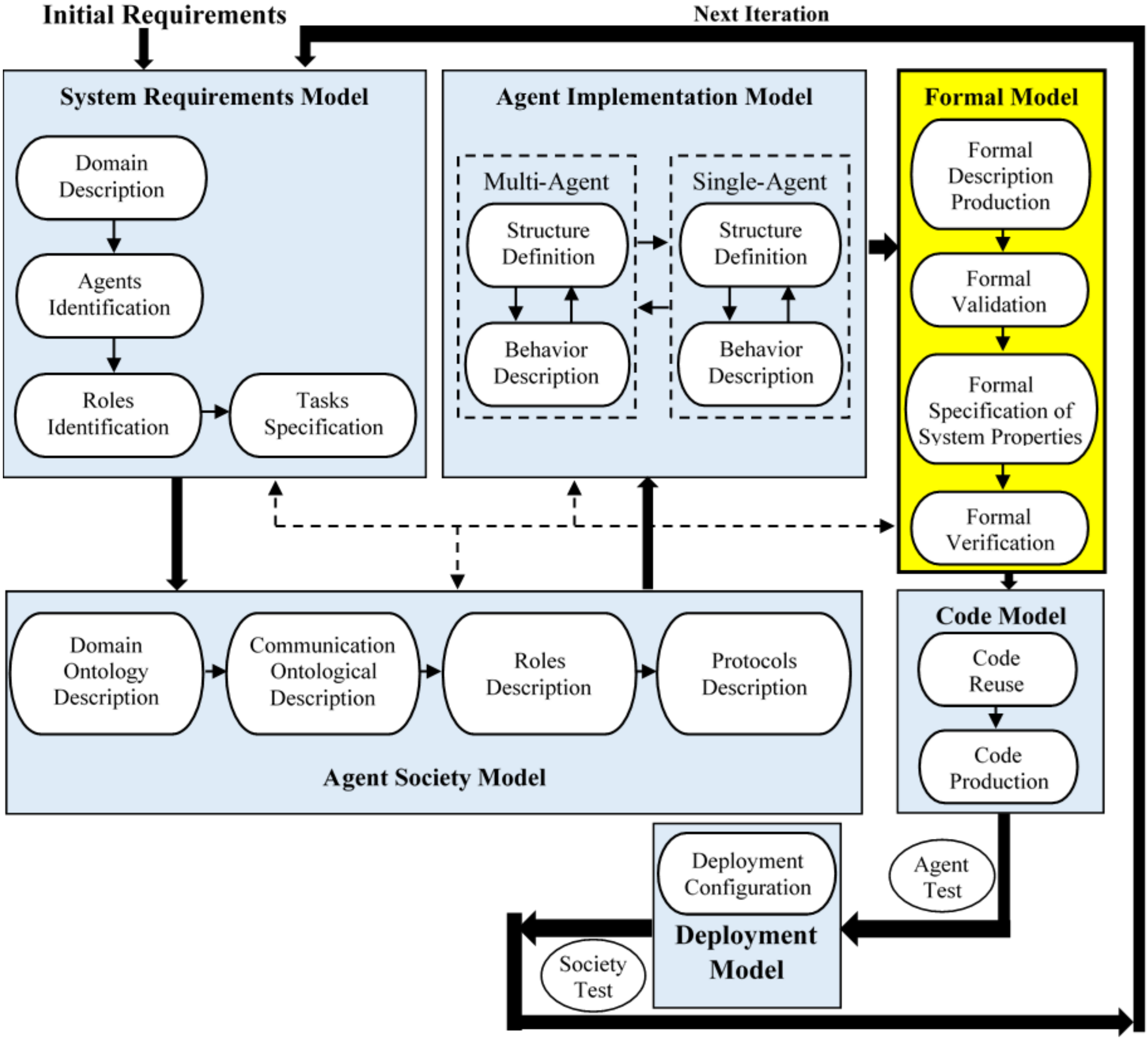


Figure 7: Formal PASSI methodology.

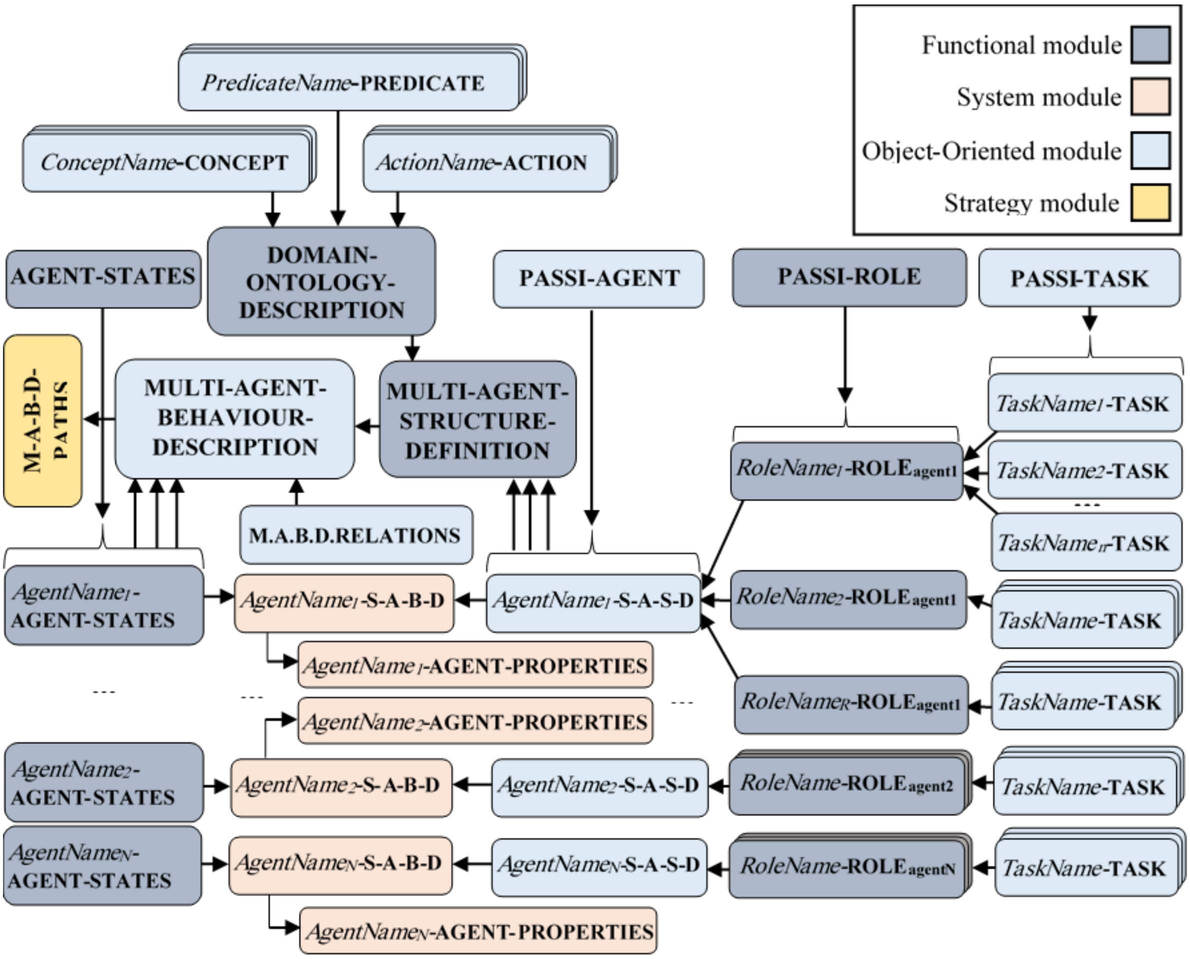


Figure 8: Generated modules.

5.1 Formal Description Production

In this phase, a Maude specification is generated from some PASSI diagrams: *Domain Ontology Description, Roles Description, Single-Agent Structure Definition, Multi-Agent Behaviour Description and Single-Agent Behaviour Description*. In the end of this phase, a Maude formal description that covers the agent's shared knowledge (domain ontology), the structure and the behaviour of the system in both multi/single abstraction levels will be available to be exploited in the next phases. The Generation is considered as a Model-to-Text transformation and automatically performed thanks to F-PTK (see section 6). Figure 8 shows the generated modules

The *Domain Ontology Description* diagram is represented formally in Maude as follows: 1) A concept having the name "*ConceptName*" is translated as a class defined in an object-oriented module with the name "*CONCEPT-NAME-CONCEPT*". 2) A predicate having the name "*PredicateName*" is translated as a class defined in an object-oriented module with the name "*PREDICATE-NAME-PREDICATE*". 3) An action having the name "*ActionName*" is translated as a class defined in an object-oriented module with the name "*ACTION-NAME-ACTION*". All the modules representing the ontology elements (concepts, predicates and actions) are imported in a functional module called "DOMAIN-ONTOLOGY-DESCRIPTION".

According to PASSI terminology, an agent-based application is composed of agents, agents play roles, roles consist of tasks and tasks consist of actions. Table.1 represents the basic concepts that PASSI methodology is based on (*TASK, ROLE and AGENT*) and their representation in Maude.

Figure 9 shows the functional module *AGENTS-STATES* which defines the sort *AgentState* representing an agent state, and defines two (2) operators: *Created* and *Initialized* representing the common states for all agents.

```
(fmod AGENT-STATES is
    sort AgentState .
    *** Commun agent states
    ops Created Initialized : -> AgentState .
endfm)
```

Figure 9: AGENTS-STATES module.

PASSI Basic Concepts	Maude Representation	Description
Task (Super class)	(omod PASSI-TASK is inc STRING . class Task | superClassTaskName : String . op noneTask : -> Cid . op noneAction : -> Msg . **** JADE commun methods for all* ****subclass Tasks* msgs action done : ParametersList -> Msg . endom)	This module defines the *Task* class that represents the task concept. As tasks will be interpreted next, in code level, by behaviours (according to JADE framework), the *superClassTaskName* attribute expresses the type of the behaviour (like, for instance, OneShotBehaviour, CyclicBehaviour). *NoneTask* and *noneAction* express the fact that the agent did not perform yet neither task nor action. The common methods: *action*, and *done* (according to JADE framework) for all subclass tasks are expressed through messages.
Role (Super class)	(fmod PASSI-ROLE is sorts Role, NextPlayedRole . op noneRole : -> Role . **** Specifying that the agent is in* ****initialization step, no role played yet* endfm)	The Role concept is represented by a functional module in which a sort called Role is defined. In this module, NextPlayedRole sort is also defined to express the [ROLE_CHANGE] relationship specified during Roles Description phase. To express that the agent didn't play any role yet, the operator *noneRole* is defined.
Agent (Super class)	(omod PASSI-AGENT is pr PASSI-ROLE . pr PASSI-TASK . pr AGENT-STATES . **** PASSI Agent class declaration* class Agent | playsRole : Role, performsTask : Task , executesTaskAction : Msg, currentState : AgentState . **** JADE commun methods for all* **** subclass agents* msgs setup registerToDF takeDown : ParametersList -> Msg . endom)	The *Agent* concept is represented by an object-oriented module in which a class called *Agent* having four (4) attributes is defined: 1) ***playsRole***: of sort Role (defined in the imported PASSI-ROLE module), signifies which role is played by the agent in a given moment. 2) ***performsTask***: of sort Task (defined in the imported PASSI-TASK module), signifies which task the agent is performing in a given moment. 3) ***executesTaskAction***: of sort Msg (predefined in Full-Maude), signifies which action the agent is executing in a given moment. 4) ***currentState***: of sort AgentState (defined in the imported AGENTS-STATES module, see Figure 9), identifies the state of the agent in a given moment among all its possible states specified in the Single Agent Behaviour Description diagram. The common methods: *setup, registerToDF* and *takedown* (according to JADE framework) for all agents subclasses are expressed through messages.

Table 1: PASSI basic concepts and their Maude representations.

A *Single Agent Structure Definition* diagram of an agent called "*AgentName*" is represented formally in Maude by an object-oriented module with the name "AGENT-NAME-SINGLE-AGENT-STRUCTURE-DEFINITION". Modules representing all the roles played by such agent, must be imported, also, all modules representing tasks composing a role must be imported in the role module. All states defined in the *Single Agent Behaviour Description* diagram for an agent *"AgentName"* are represented in a functional module with the name "AGENT-NAME-AGENT-STATES".

As mentioned in [2], *TaskActions* in *Multi-Agent Behaviour Description* diagram are related by *Invocation, Done, NewTask and Message* relationships. These relationships are defined in the *M-A-B-D-RELATIONSHIPS* (see Figure 10). Besides these relationships, the module defines types as: *OntologyElement* that can be a concept, a predicate or an action; *Performative* that signifies the communication performative mentioned in a *message* relationship. In order to express that a *Task* has been instantiated, a *TaskAction* has been executed and that a message has been sent, we have defined respectively three messages *TaskInstantiated, TaskActionExecuted and MessageSent*. The *FinalState* message expresses a final state of a scenario.

The *Multi-Agent Behaviour Description* diagram is translated in Maude by the object-oriented module "MULTI-AGENT-BEHAVIOUR-DESCRIPTION". In this module, all modules representing the structure of agents as well as all functional modules representing their states in addition to the *M-A-B-D-RELATIONSHIPS* module are imported.

All execution paths of MABD diagram are automatically captured (thanks to F-PTK tool) and represented as strategies thanks to Maude-Strategy in a strategic module with the name "MULTI-AGENT-BEHAVIOUR-DESCRIPTION-PATHS".

```
(omod M-A-B-D-RELATIONS is
 inc CONFIGURATION .
 inc STRING .
 pr PASSI-ROLE .
 pr PASSI-TASK .
 sorts OntologyElement Performative .
 subsort Cid < OntologyElement .
 subsort String < Performative .
 sorts Initiator Participant .
 subsort Cid < Initiator .
 subsort Cid < Participant .
   *** Relations among task actions
 msgs invocation Done : Msg -> Msg .
   *** <Task class name> Relation among tasks>
 msg newTask : Task -> Msg .
   *** <OntologyElement class name>
 msg message : OntologyElement Performative -> Msg .
   *** Action Task Agent
 msg TaskActionExecuted : Msg Cid Cid -> Msg .
   *** Task Role Agent
 msg TaskInstantiated : Cid Role Cid -> Msg .
    msg MessageSent : Initiator Participant
           OntologyElement Performative -> Msg .
    msg FinalState : ParametersList -> Msg .
endom)
```

Figure 10: M-A-B-D-RELATIONS module.

Despite the many potential benefits that formal specifications offer, they suffer from two major limits, scalability and familiarity of the developers using them with the logics/languages on which the formal methods are based on. The first limit pushes the researchers' community to make formal methods applicable not only to small-scale applications but also to large-scale applications. The second means that the developers using formal methods need to have a high degree of mathematical maturity as well as languages the formal methods they use are based on. To overcome these limits in our approach, we have developed a tool, Formal-PASSI Toolkit (see section 6). In one hand, the developed tool should contribute to scale up our approach. In the other hand, it limits the intervention of developers in the specification of properties to be checked, and let them deal with the semi-formal notation (UML notation) that PASSI is based on.

5.2 Formal Validation

The particularity of the generated formal description, knowing that it is developed using objects, messages, and rewriting rules, is that it is executable. As Maude is a very versatile environment in terms of simulation, it is possible to define a customized initial state (initial configuration) and to execute this configuration of the system. Two diagrams (until now) are considered by the validation: Single and Multi-Agents Behaviour Description diagrams. For the first one, the validation process begins by introducing one or more initial configuration(s) of an agent with its knowledge (ontology elements). For the second diagram (MABD diagram), the validation process begins by introducing one or more initial configuration(s) composed of all agents and the knowledge they need. After executing the simulation, the developer has to read the obtained results from the given initial configuration(s) and judge if it is expected or not. If the given result(s) is (are) undesirable(s), he should take a look to the SABD diagram and/or MABD diagram for a certain modification.

5.3 Formal Specification of System Properties

In this phase, the designer (that is supposed to be familiar with Linear Temporal Logic and Maude language) has to specify formally some properties (desirables or not) of the MAS (Multi-agent abstract level) and of individual agents (Single-agent abstract level) to be checked in the next phase. A list of properties related specifically to multi-agent systems will be the subject of a future paper. For that, as a starting point, all states of an agent "AgentName1" should be specified as elementary predicates in a system module "AGENT-NAME1-PREDICATES". Since a MAS is composed of agents, a property of a MAS is constructed by the composition of elementary predicates (each of them expresses an agent in one of its states) via LTL operators.

5.4 Formal Verification

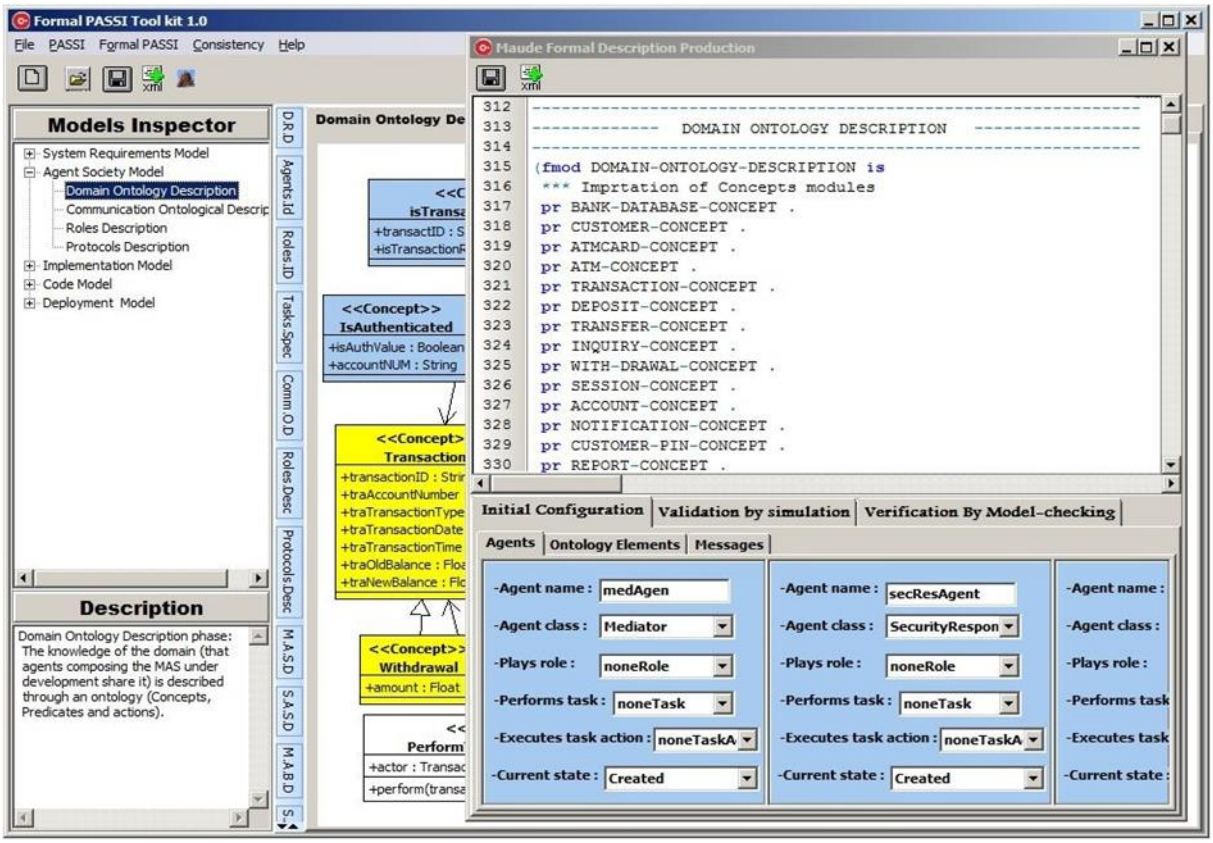
During this phase, a model checking of some PASSI behavioural diagrams is performed. Model checking aims at applying an exhaustive analysis of all possible execution paths of a system, and to determine if some properties (identified in the previous phase) are satisfied or not. Applying this technique on the formal description, generated previously, is very important to verify Multi/Single-A-B-D diagrams. This would have the advantage of applying model checking before passing to Code Model and to avoid propagation of subtle errors introduced at the level of the three models (System requirement model, Agent society model and Implementation Model), with the remainder of the development process (Code Model, Agent Test activity, Deployment Model and Society Test activity).

6 Formal PASSI Toolkit

To make F-PASSI valid and its adoption wider by researchers (and possibly industry with more big dimension MASs), we must offer users the tool(s) to support it. For that, we have developed a prototype toolkit, F-PTK (Formal PASSI-Toolkit), using C++ Builder XE7[1] IDE. Figure 11 shows the developed toolkit.

Among the options offered by F-PTK in its version 1.0, we mention: (1) Edit the different PASSI diagrams, (2) Detect automatically the different paths defined in the Multi-Agent Behaviour Description diagram and translate it as Maude-Strategies to be used in the formal validation phase, (3) Check the consistency of the diagrams, (4) Serialize these diagrams for later use (to XML file), (5)

[1] https://www.embarcadero.com/fr/products/cbuilder



Domain Ontology Description phase: The knowledge of the domain (that agents composing the MAS under development share it) is described through an ontology (Concepts, Predicates and actions).

```
------------------------------------------------------------------------
-------------   DOMAIN ONTOLOGY DESCRIPTION   ------------------
------------------------------------------------------------------------
(fmod DOMAIN-ONTOLOGY-DESCRIPTION is
 *** Imprtation of Concepts modules
 pr BANK-DATABASE-CONCEPT .
 pr CUSTOMER-CONCEPT .
 pr ATMCARD-CONCEPT .
 pr ATM-CONCEPT .
 pr TRANSACTION-CONCEPT .
 pr DEPOSIT-CONCEPT .
 pr TRANSFER-CONCEPT .
 pr INQUIRY-CONCEPT .
 pr WITH-DRAWAL-CONCEPT .
 pr SESSION-CONCEPT .
 pr ACCOUNT-CONCEPT .
 pr NOTIFICATION-CONCEPT .
 pr CUSTOMER-PIN-CONCEPT .
 pr REPORT-CONCEPT .
```

-Agent name : medAgen
-Agent class : Mediator
-Plays role : noneRole
-Performs task : noneTask
-Executes task action : noneTaskA
-Current state : Created

-Agent name : secResAgent
-Agent class : SecurityRespon
-Plays role : noneRole
-Performs task : noneTask
-Executes task action : noneTaskA
-Current state : Created

-Agent name :
-Agent class :
-Plays role :
-Performs task
-Executes task
-Current state :

Figure 11: Formal PASSI Toolkit1.0.

Generate the Maude-based formal description of the MAS, (6) Save the generated formal description as an XML file, (7) Validate the generated description, and (8) Verify the generated description after giving the properties to be checked

In addition to the fact that F-PTK supports Formal-PASSI, it is characterized mainly from PTK [58] by being based on our proposed traceability meta-model for PASSI methodology [60]. This will guide developers when designing the different diagrams and facilitate theirs tasks.

7 Case study

Our proposed extension will be made concrete and illustrated using the ATM (Automated Teller Machine) case study. The MAS to be designed will control a simulated automated teller machine having a magnetic stripe reader to read an ATM card, a customer console to interact with customers, a slot to deposit envelopes, a dispenser for cash and a printer to print customer receipts. A customer should insert an ATM card and enter a PIN (Personal Identification Number). The Card information and the entered PIN will be sent to the bank for validation before each session. After validating the customer's card and PIN, the customer will then be able to perform one or more transactions. The customer could regain its card when he/she desires no further transaction, or when he/she decides to abort the transaction in progress. The designed ATM provides the following basic services: (1) Perform a cash withdrawal from the account related to the inserted card; (2) Perform a deposit to any account related to the inserted card; (3) Perform a transfer of money between any two accounts linked to the inserted card; (4) Perform a balance inquiry of any account related to the inserted card; (5) Abort a transaction in progress if the "Cancel" key is pressed by the customer.

7.1 Our design of the ATM case study through PASSI

In this section, we show our own design of the ATM case study described above. For space limitation reasons, only some of the diagrams adopted in our formalization approach (until now) are showed or discussed.

Agents Identification (AI): Three agents are identified: (1) Mediator Agent: It is responsible of displaying information on the ATM screen (about ATM available options, information after a successful transaction, etc.), reading customer's ATM card. (2) Transaction Manager Agent: It is responsible of performing transactions, reporting transactions, printing receipts for successful transaction. (3) Security Responsible Agent: It is responsible of checking customer's card, authenticating the customer, ensuring privacy when a transaction is in progress.

Domain Ontology Description (DOD): In this step, the knowledge of the domain is described from an ontological perspective. For example, the concept "Transaction" is identified with its identifier, its date, its time, etc. The fact of being "withdrawal", "Inquiry", Transfer" and "Deposit" kinds of transaction, this made them identified as concepts inheriting the "Transaction" concept. Also, the predicate “IsTransactionPerformed” is identified to know if the transaction is successfully performed (isTransPerfValue=true) or not (isTransPerfValue=false).

Roles Description (RD): In this step, the roles played by agents are packaged (see Figure 12). For the “TransactionManager” agent, two roles are identified: “Performer” which represents the case in which the agent is performing a transaction, “Reporter” which describes the case in which the agent is reporting a transaction. Whereas, three roles are identified for both of “SecurityResponsible” and “Mediator” agents. “AccountChecker”, “Authenticator” and “Saver” for the first one, “CardReader”, “AmountChecker” and “Dispenser” for the last.

Single-Agent Structure Definition (SASD): The “TransactionManager” agent has ten tasks to perform when playing its roles. Among them, we mention for example, the “AbortTransaction” task, which is performed when the ATM customer presses the “Cancel” button to abort the transaction in progress. However, the “AskForDispensing” task, is performed to ask the “Mediator” agent to dispense the customer’s desired amount.

Multi-Agent Behaviour Description (MABD): Figure 13 shows a part of the Multi-Agent Behaviour Description diagram we design for our case study. The figure shows how task actions are executed, the different messages sent between different agents or tasks. For example, the message (Notification, Inform) is sent by the “Security-Responsible” agent (its “sendReportNotification” task action) to the “Transaction-Manager” agent (its “receiveReportNotification” task action).

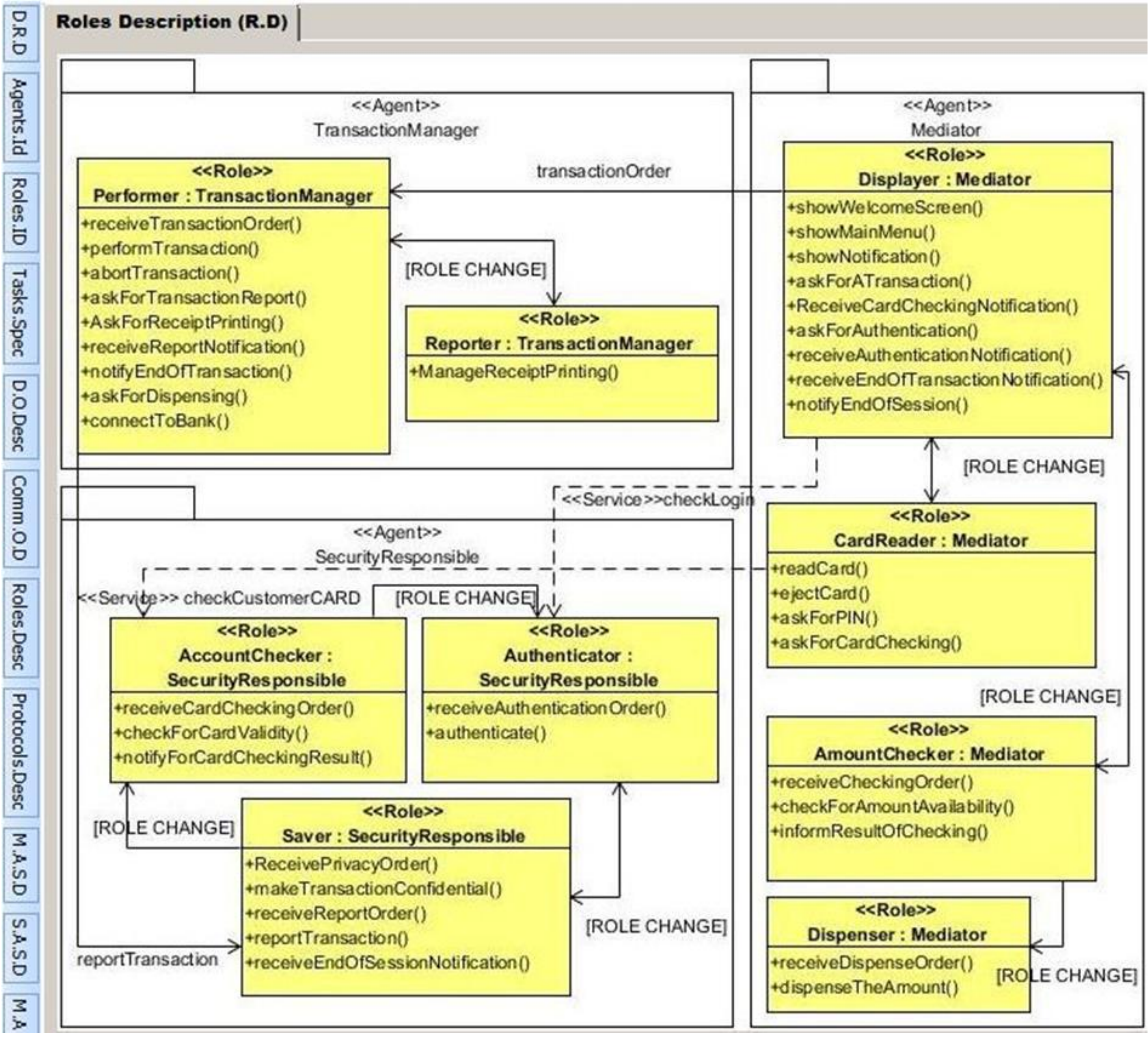


Figure 12: Roles Description diagram of ATM case study.

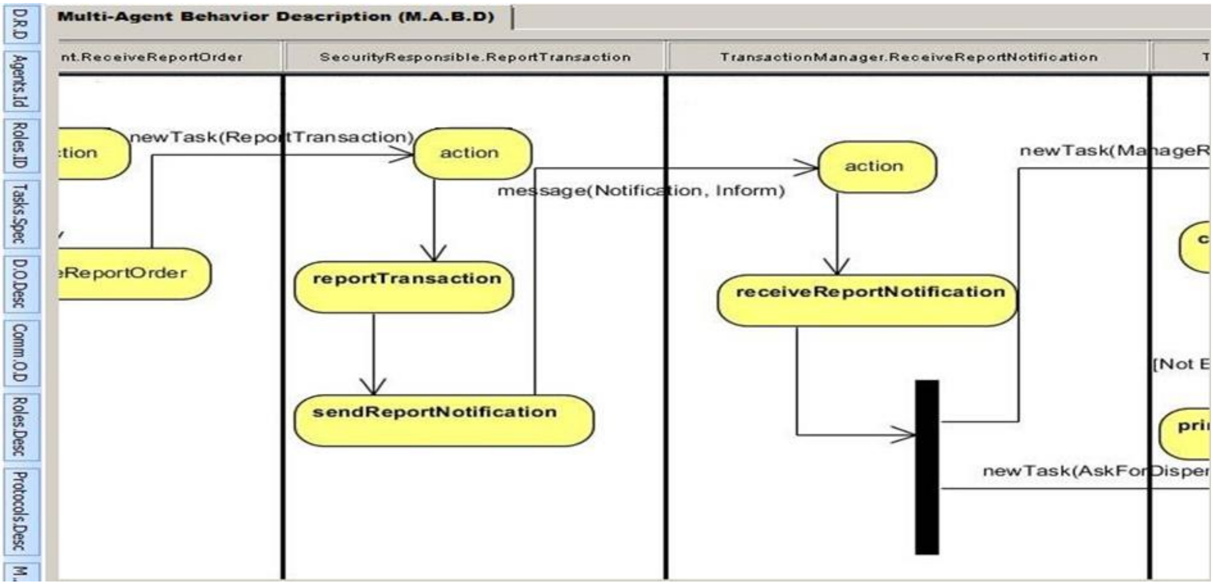


Figure 13: A part of Multi-Agent Behaviour Description diagram of ATM case.

Single-Agent Behaviour Description diagram: Figure 14 shows a finite state machine representing the behaviour of the agent "*Transaction Manager*". We identified twelve (12) states for this agent. For example, after asking the Mediator agent for dispensing money ("AskingFor-MoneyDispensing" state), the TransactionManager agent will be in the "NotifyingForTransactionEnd" state by executing "notifyEndOfTransaction(aNotification : Notification)" task action.

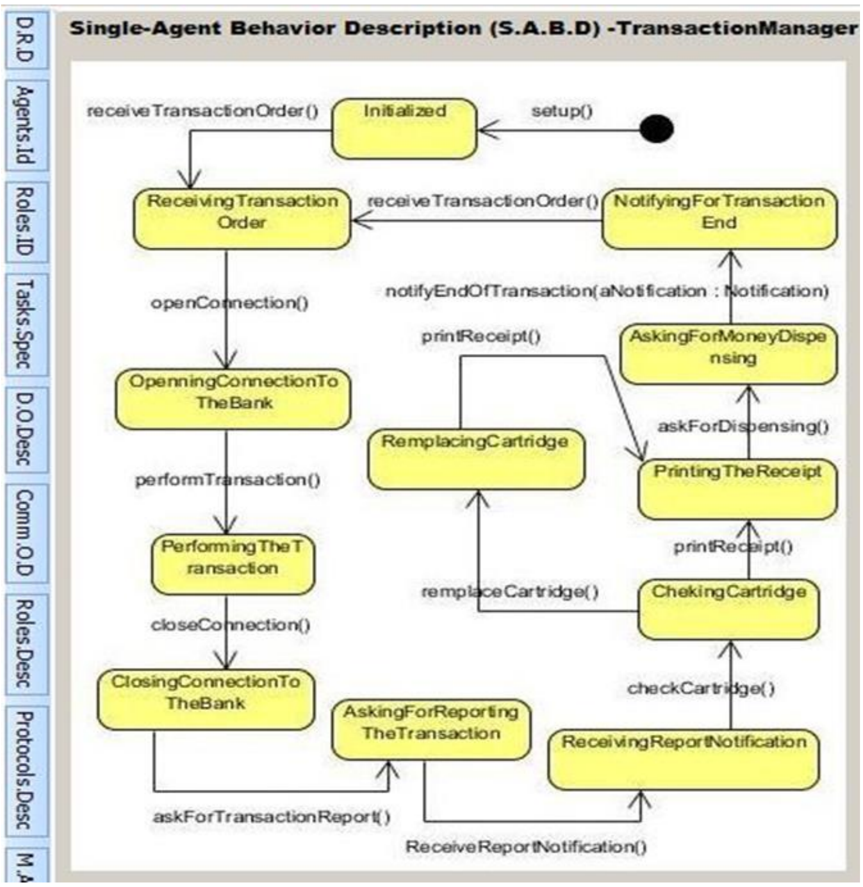

D.R.D
Agents.Id
Roles.ID
Tasks.Spec
D.O.Desc
Comm.O.D
Roles.Desc
Protocols.Desc
M.A
Single-Agent Behavior Description (S.A.B.D) -TransactionManager
receiveTransactionOrder()
Initialized
setup()
ReceivingTransaction Order
receiveTransactionOrder()
NotifyingForTransaction End
openConnection()
notifyEndOfTransaction(aNotification : Notification)
printReceipt()
AskingForMoneyDispensing
OpenningConnectionTo TheBank
RemplacingCartridge
askForDispensing()
PrintingTheReceipt
performTransaction()
PerformingTheTransaction
printReceipt()
remplaceCartridge()
ChekingCartridge
closeConnection()
ClosingConnectionTo TheBank
checkCartridge()
AskingForReporting TheTransaction
ReceivingReportNotification
askForTransactionReport()
ReceiveReportNotification()

Figure 14: Single-Agent Behaviour Description diagram of ATM case study -Transaction Manager-Agent-.

7.2 Formal Model for ATM

Formal Description Production

Using F-PTK, a Maude specification of the MAS under development is produced. As we have mentioned before, the domain ontology elements (concepts, predicates and actions) are translated in Maude as classes defined in object-oriented modules. The following figure (Figure 15) shows the corresponding Maude-representation of the “IsAuthenticated” predicate. The attribute “isAuthValue” of Boolean type, expresses if the customer having the account “accountNum” is authenticated or not.

```
 (omod IS-AUTHENTICATED-PREDICATE is
   pr BOOL .
   pr STRING .
   class IsAuthenticated | isAuthValue : Bool,
                           accountNUM : String .
 endom)
```

Figure 15: ”IsAuthenticated” predicate in Maude.

The structure of the “*TransactionManager*” agent is represented in Maude, as Figure 16 shows, by an object-oriented module. A class with the same name as the agent’s name is defined (line: *979*). This class (as any other agent’s class) has to inherit (line: *980*) the “*Agent*” class (defined in AGENT-PASSI module, see Table.1). The roles played by this agent (Performer and Reporter) are captured from Roles Description diagram, and the modules in which they are defined are imported (lines: *973,974*) as well as the module representing the domain ontology (line: *976*).

```
(omod TRANSACTION-MANAGER-SINGLE
        -AGENT-STRUCTURE-DEFINITION is
  pr PASSI-AGENT .  *** line 970
  pr MESSAGE .
  *** Roles modules importation
  pr PERFORMER-ROLE . *** line 973
  pr REPORTER-ROLE . *** line 974
  *** The "Domain Ontology Description"
 *** module importation
  pr DOMAIN-ONTOLOGY-DESCRIPTION .*** line 976
  *** Modules importation for different Maude types
  *** Agent class declaration
  class TransactionManager | transaction : Oid .*** line 979
  subclass TransactionManager < Agent . *** line 980
endom)
```

Figure 16: Single-Agent Structure Definition module of the Agent Transaction Manager.

Figure 17 demonstrates the module representing the *MULTI-AGENT-BEHAVIOUR-DESCRIPTION* diagram. This module imports the following modules: *1) MULTI-AGENT-STRUCTURE-DEFINITION module (line: 1014). 2) M-A-B-D-RELATIONSHIPS module (line: 1016). 3) All modules representing the states of agents composing the MAS (lines: 1018, 1019 and 1020).*

```
(fmod MULTI-AGENT-STRUCTURE-DEFINITION is
 inc MEDIATOR-SINGLE-AGENT-STRUCTURE-DEFINITION .
 inc SECURITY-RESPONSIBLE-SINGLE-AGENT-
     STRUCTURE-DEFINITION .
 inc TRANSACTION-MANAGER-SINGLE-AGENT-
      STRUCTURE-DEFINITION .
endfm)
****************************
(omod MULTI-AGENT-BEHAVIOUR-DESCRIPTION is
 inc MULTI-AGENT-STRUCTURE-DEFINITION .***line : 1014
  ***
  inc M-A-B-D-RELATIONS .  ***line : 1016
  ***
  inc MEDIATOR-AGENT-STATES .   ***line : 1018
  inc TRANSACTION-MANAGER-AGENT-STATES .***line: 1019
  inc SECURITY-RESPONSIBLE-AGENT-STATES . ***line : 1020
  …
endom)
```

Figure 17: *MULTI-AGENT-STRUCTURE-DEFINITION* module and a part of *MULTI-AGENT-BEHAVIOUR-DESCRIPTION* module.

All relationships relating *Task Actions* appearing in the *Multi-Agent Behaviour Description* diagram are translated as rewriting rules. The execution of each rewriting rule affects the agents’ states and the used ontology elements.

Figure.18 shows a rewriting rule (labelled by: MABD-35, line: 1433) which represents the execution of the task action “notifiyForAuthenticationResult” in the case of invalid PIN entered. In which case, the “Security Responsible” agent’s state is changed from “AuthenticatingTheCustomer” to “SendingAuthenticationResult” (lines: 1438 and 1446), also, the predicate object “IsAuthent” with the value false, and a notification object “notif” with the content “Your Pin is invalid, please enter a correct one” are generated (lines: 1441,1442-1443).

Figure.19 shows a part of strategies captured from the MABD diagram and defined in the strategic module MULTI-AGENT-BEHAVIOUR-DESCRIPTION-PATHS.

Formal Validation

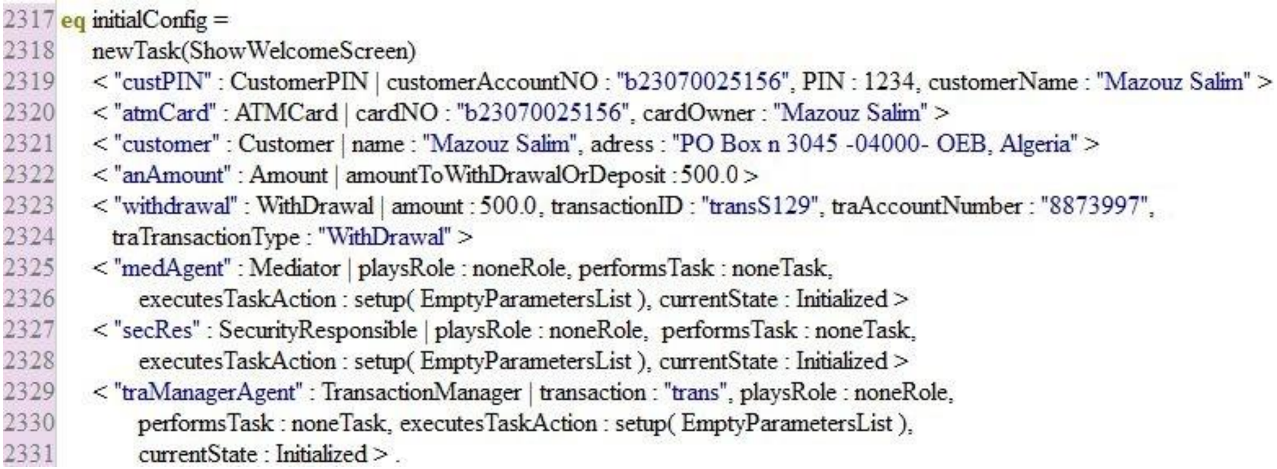
Once the Maude-based formal description of the MAS is generated, a formal validation by simulation becomes possible. Figure 20 shows an initial configuration in which a customer called “*Mazouz Salim*” (line: *2321*) chooses to perform a withdraw transaction of an amount of : € 500,00 (line: *2323*). The three (3) agents are, in first time, initialized (lines: 2326, 2328 and 2331).

```
rl[ MABD-35 ] :  *** line :1433
    invocation(notifyForAuthenticationResult(notif))
    < custPIN : CustomerPIN | customerAccountNO : accno, PIN : pin, customerName : custName >
    < secRes: SecurityResponsible | playsRole : Authenticator, performsTask : Authenticate,
                                  executesTaskAction : authenticate(EmptyParametersList),
                                  currentState: AuthenticatingTheCustomer > ***line : 1438
    =>
    message(Notification, "Inform")
    < “isAuthent” : IsAuthenticated | isAuthValue : false, accountNUM : accno > ***line : 1441
    < “notif” : Notification | notifID : notID,          ***line : 1442
                          content : "Your Pin is invalid, please enter a correct one" > ,  ***line : 1443
    < secRes : SecurityResponsible | playsRole : Authenticator, performsTask : Authenticate,
                                  executesTaskAction : notifyForAuthenticationResult(notif),
                                  currentState : SendingAuthenticationResult > . ***line : 1446
```

Figure 18: A rewriting rule of the MABD module.

```
      (smod MULTI-AGENT-BEHAVIOUR-DESCRIPTION-PATHS is
        strat Root : @ Configuration .
        sd Root := ( MABD-01 ; MABD-02 ; MABD-03 ; MABD-04 ;
                     MABD-05 ; MABD-06 ; MABD-07 ; MABD-08 ;
                     MABD-09 ).
         …
        strat Parall1-1 : @ Configuration .
        sd Parall1-1 := ( Branch1-6 | Branch1-7 )! .
        *** Case of well passed scenario
        strat Path1 : @ Configuration . *** line 2215
        sd Path1 := ( Root ; Branch1-1 ; Branch1-2 ; Branch1-3 ;  *** line 2216
                      Branch1-4 ;  Branch1-5 ;  Parall1-1 ; Branch1-8 ) .  *** line 2217
        …
      endsm)
```

Figure 19: A part of the strategic module representing the different paths of Multi-Agent Behaviour Description diagram.



```
eq initialConfig =
   newTask(ShowWelcomeScreen)
   < "custPIN" : CustomerPIN | customerAccountNO : "b23070025156", PIN : 1234, customerName : "Mazouz Salim" >
   < "atmCard" : ATMCard | cardNO : "b23070025156", cardOwner : "Mazouz Salim" >
   < "customer" : Customer | name : "Mazouz Salim", adress : "PO Box n 3045 -04000- OEB, Algeria" >
   < "anAmount" : Amount | amountToWithDrawalOrDeposit : 500.0 >
   < "withdrawal" : WithDrawal | amount : 500.0, transactionID : "transS129", traAccountNumber : "8873997",
     traTransactionType : "WithDrawal" >
   < "medAgent" : Mediator | playsRole : noneRole, performsTask : noneTask,
        executesTaskAction : setup( EmptyParametersList ), currentState : Initialized >
   < "secRes" : SecurityResponsible | playsRole : noneRole,  performsTask : noneTask,
        executesTaskAction : setup( EmptyParametersList ), currentState : Initialized >
   < "traManagerAgent" : TransactionManager | transaction : "trans", playsRole : noneRole,
        performsTask : noneTask, executesTaskAction : setup( EmptyParametersList ),
        currentState : Initialized > .
```

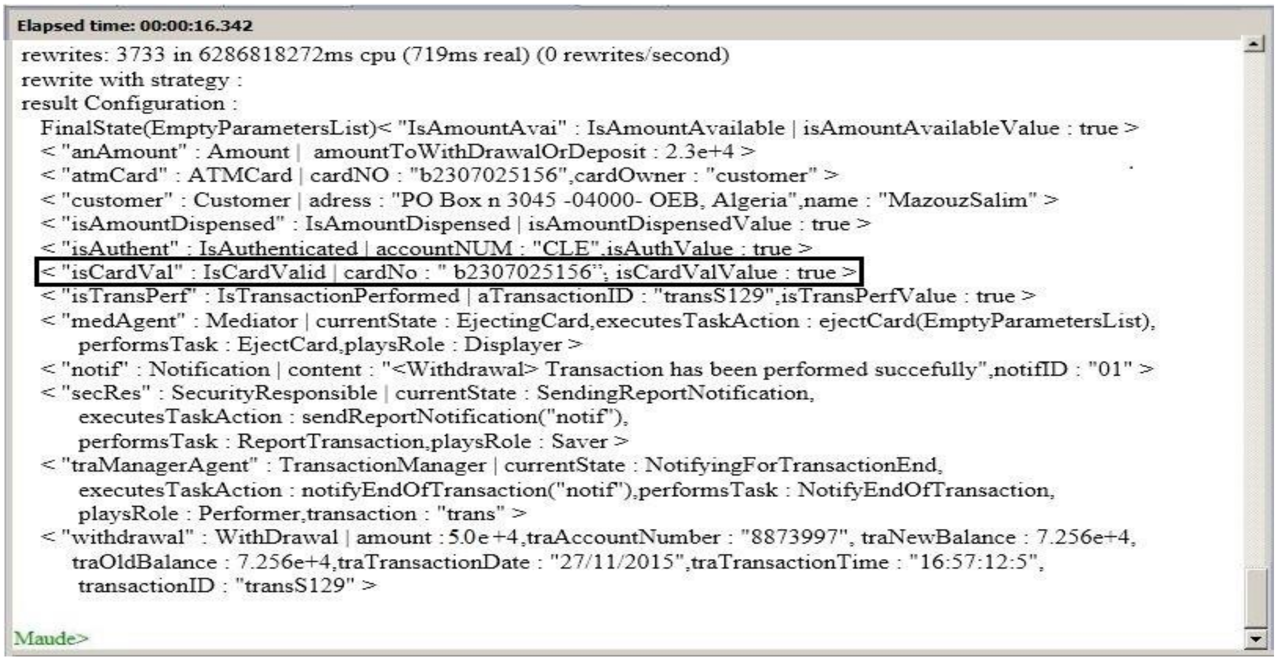
Figure 20: An initial configuration.

The results of simulating the initial configuration (Figure.20) by executing the strategy “Path1” (see *lines: 2215, 2216 and 2217* in Figure 19) are showed in Figure 21. This strategy illustrates the case, in which, the inserted customer’s card was valid, the customer has been authenticated and the withdrawal transaction has been successfully performed.

The results of this phase gives the developer more information about agents by means of their states, the roles they played, tasks they performed, and task actions they executed in addition to the current values of ontology element’s attributes. For example, the predicate “isCardVal” (framed by the black in Figure 21) gives us the information that the card inserted by the customer was valid. <"isCardVal" : IsCardValid | cardNo : "b2307025156", isCardValValue : true>.

Formal Specification of System Properties

In this phase, some of MAS properties are identified and then specified in Linear Temporal Logic as predicates in Maude. Table 2 shows some properties for the ATM case study and their specification in LTL logic. Three of these properties (desirable properties) should be satisfied by the MAS, whilst, the others are undesirable and the MAS should not satisfy them.



```
Elapsed time: 00:00:16.342
rewrites: 3733 in 6286818272ms cpu (719ms real) (0 rewrites/second)
rewrite with strategy :
result Configuration :
  FinalState(EmptyParametersList)< "IsAmountAvai" : IsAmountAvailable | isAmountAvailableValue : true >
  < "anAmount" : Amount |  amountToWithDrawalOrDeposit : 2.3e+4 >
  < "atmCard" : ATMCard | cardNO : "b2307025156",cardOwner : "customer" >
  < "customer" : Customer | adress : "PO Box n 3045 -04000- OEB, Algeria",name : "MazouzSalim" >
  < "isAmountDispensed" : IsAmountDispensed | isAmountDispensedValue : true >
  < "isAuthent" : IsAuthenticated | accountNUM : "CLE",isAuthValue : true >
  < "isCardVal" : IsCardValid | cardNo : " b2307025156"; isCardValValue : true >
  < "isTransPerf" : IsTransactionPerformed | aTransactionID : "transS129",isTransPerfValue : true >
  < "medAgent" : Mediator | currentState : EjectingCard,executesTaskAction : ejectCard(EmptyParametersList),
      performsTask : EjectCard,playsRole : Displayer >
  < "notif" : Notification | content : "<Withdrawal> Transaction has been performed succefully",notifID : "01" >
  < "secRes" : SecurityResponsible | currentState : SendingReportNotification,
      executesTaskAction : sendReportNotification("notif"),
      performsTask : ReportTransaction,playsRole : Saver >
  < "traManagerAgent" : TransactionManager | currentState : NotifyingForTransactionEnd,
      executesTaskAction : notifyEndOfTransaction("notif"),performsTask : NotifyEndOfTransaction,
      playsRole : Performer,transaction : "trans" >
  < "withdrawal" : WithDrawal | amount : 5.0e +4,traAccountNumber : "8873997", traNewBalance : 7.256e+4,
     traOldBalance : 7.256e+4,traTransactionDate : "27/11/2015",traTransactionTime : "16:57:12:5",
      transactionID : "transS129" >

Maude>
```

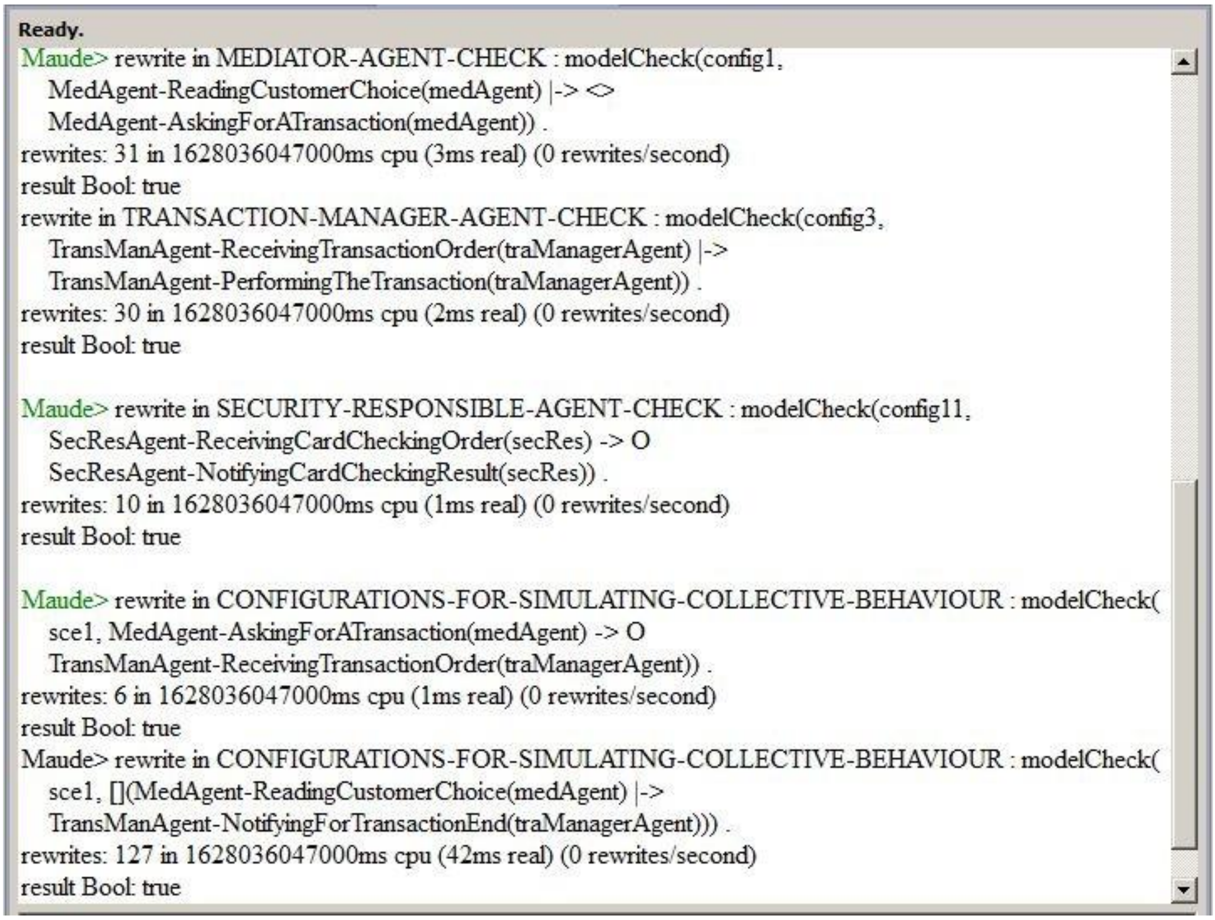
Figure 21: Result of the simulation (Scenario well passed)
By the command (srew initialConfig using Path1 .)

N°	Property in LTL	Description	Desirable Property	Single/Multi-Agent
1	MedAgent-ReadingCustomerChoice(medAgent) |-> <> MedAgent-AskingForATransaction(medAgent)	This property expresses the fact that if the Mediator Agent reads the customer choice (ReadingCustomer-Choice state), then it will eventually send a transaction order to the TransactionManager agent soon.	Yes	Single-Agent: Mediator
2	TransManAgent-ReceivingTransactionOrder(transactionMan) |-> TransManAgent-PerformingTheTransaction(transactionMan)	This property expresses the fact that if the TransactionManager Agent receives a transaction order (ReceivingTransactionOrder state), then the state Performing-TheTransaction expressing that it is performing the transaction will be true soon.	Yes	Single-Agent: TransactionManager
3	SecResAgent-ReceivingCardCheckingOrder(secRes) -> O SecResAgent-NotifyingCardCheckingResult (secRes)	This property expresses the fact that if the Security Responsible agent receives a card checking order, it will notify directly the results of checking (without checking it first).	No	Single-Agent: Security Responsible
4	MedAgent-AskingForATransaction(medAgent) -> O TransManAgent-ReceivingTransactionOrder(traManagerAgent)	This property expresses the fact that if the Mediator agent sends a transaction order to the TransactionManager agent, the last one will be in ReceivingTransactionOrder state.	Yes	Multi-Agent
5	[] (MedAgent-ReadingCustomerChoice(medAgent) |-> TransManAgent-NotifyingForTransactionEnd(traManagerAgent))	This property expresses the situation: Always, if the MediatorAgent is reading the customer choice, the TransactionManager agent will notify for the end of the transaction.	No	Multi-Agent

Table 2: **S**ome specified properties for the ATM case study.

Formal Verification

After the specification of properties (Table 2), a verification by means of model checking technique is applied. Figure 22 illustrates the results given by Maude-Model checker on different initial configurations. In the case of desirable properties unsatisfied or undesirable properties satisfied (like the third and the fifth properties in the table above), the developer has to review the corresponding diagrams for modification.



```
Ready.
Maude> rewrite in MEDIATOR-AGENT-CHECK : modelCheck(config1,
   MedAgent-ReadingCustomerChoice(medAgent) |-> <>
   MedAgent-AskingForATransaction(medAgent)) .
rewrites: 31 in 1628036047000ms cpu (3ms real) (0 rewrites/second)
result Bool: true
rewrite in TRANSACTION-MANAGER-AGENT-CHECK : modelCheck(config3,
   TransManAgent-ReceivingTransactionOrder(traManagerAgent) |->
   TransManAgent-PerformingTheTransaction(traManagerAgent)) .
rewrites: 30 in 1628036047000ms cpu (2ms real) (0 rewrites/second)
result Bool: true

Maude> rewrite in SECURITY-RESPONSIBLE-AGENT-CHECK : modelCheck(config11,
   SecResAgent-ReceivingCardCheckingOrder(secRes) -> O
   SecResAgent-NotifyingCardCheckingResult(secRes)) .
rewrites: 10 in 1628036047000ms cpu (1ms real) (0 rewrites/second)
result Bool: true

Maude> rewrite in CONFIGURATIONS-FOR-SIMULATING-COLLECTIVE-BEHAVIOUR : modelCheck(
   sce1, MedAgent-AskingForATransaction(medAgent) -> O
   TransManAgent-ReceivingTransactionOrder(traManagerAgent)) .
rewrites: 6 in 1628036047000ms cpu (1ms real) (0 rewrites/second)
result Bool: true
Maude> rewrite in CONFIGURATIONS-FOR-SIMULATING-COLLECTIVE-BEHAVIOUR : modelCheck(
   sce1, [](MedAgent-ReadingCustomerChoice(medAgent) |->
   TransManAgent-NotifyingForTransactionEnd(traManagerAgent))) .
rewrites: 127 in 1628036047000ms cpu (42ms real) (0 rewrites/second)
result Bool: true
```

Figure 22: Results of applying model checking.

8 Conclusion and future work

Several methodologies supporting MAS development have been proposed in the literature. Only few of them have addressed the use of formal techniques in the development process. Despite the fact that PASSI methodology have many advantages such as the coverage of most development phases, the design of FIPA-based MASs[1], the use of the common modelling language (UML) and the plenty of documentations (Web site[2], lots of published papers), it lacks formal foundations. In this paper, we have presented an extension for PASSI methodology to support formal development of MAS. The extension is made by integrating a new model (*Formal Model*) into the PASSI design process. The integrated model is based on the rewriting logic and its language Maude (and its extension Maude-Strategy). It aims at offering a Maude specification of the MAS under development. Having the formal specification gives the developer the possibility to validate by simulation (thanks to Maude) of both single & multi-agent behaviour descriptions. In addition, some properties (of both single & multi-agent abstract levels) have to be specified in Maude by the developer to check it by LTL Maude model checker. Unlike many works in the literature, our work consists of integrating formal techniques not only in some design pieces separately of any development methodology, but in an entire design process (of PASSI methodology). This integration enhances PASSI methodology and leads, at the end of the design process, to the development of more reliable, robust and correct MASs. Moreover, supporting Formal PASSI by a tool (F-PTK) facilitates the tasks of the developer and would contribute to scale up our approach. Formal PASSI uses formal (rewriting logic-based) and semi-formal (UML notation) specifications, this benefits of the advantages of the two specifications. Our work is still in progress. As a future work, we plan to: (1) Introduce more PASSI diagrams in the formalization approach, (2) Formalize PASSI's predefined patterns using Maude, (3) Define and check MAS specific properties, (4) Enhance the F-PTK by adding the possibility of visualizing and animating the

[1] http://www.fipa.org/resources/methodologies.html

[2] http://www.pa.ica r.cnr.it/passi/Passi/PassiIndex.html

Formal Validation results to make them more readable, (5) Propose (or use) a graphical notation to describe LTL's operators in order to facilitate the *Formal Specification of System Properties* phase for developers who are not familiar with LTL.

References

[1] Cossentino, M. (2005). *From requirements to code with the PASSI methodology*. In B. Henderson-Sellers & P. Giorgini (Eds.), *Agent-oriented methodologies*. Hershey, PA, USA: Idea Group Publishing: Chap. IV, pp. 79–106.

[2] Cossentino, M. and V. Seidita (2014). *PASSI: Process for Agent Societies Specification and Implementation*. Handbook on Agent-Oriented Design Processes. M. Cossentino, V. Hilaire, A. Molesini and V. Seidita, Springer Berlin Heidelberg: 287-329.

[3] Cernuzzi, L., T. Juan, L. Sterling and F. Zambonelli (2004). *The Gaia Methodology*. Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US: 69-88.

[4] Cernuzzi, L., A. Molesini and A. Omicini (2014). *The Gaia Methodology Process*. Handbook on Agent-Oriented Design Processes. M. Cossentino, V. Hilaire, A. Molesini and V. Seidita. Berlin, Heidelberg, Springer Berlin Heidelberg: 141-172.

[5] Bernon, C., M.-P. Gleizes, S. Peyruqueou and G. Picard (2003). *ADELFE: A Methodology for Adaptive Multi-agent Systems Engineering*. Engineering Societies in the Agents World III: Third International Workshop, ESAW 2002 Madrid, Spain, September 16–17, 2002 Revised Papers. P. Petta, R. Tolksdorf and F. Zambonelli. Berlin, Heidelberg, Springer Berlin Heidelberg: 156-169.

[6] Bonjean, N., W. Mefteh, M. P. Gleizes, C. Maurel and F. Migeon (2014). *ADELFE 2.0*. Handbook on Agent-Oriented Design Processes. M. Cossentino, V. Hilaire, A. Molesini and V. Seidita. Berlin, Heidelberg, Springer Berlin Heidelberg: 19-63.

[7] Mefteh, W., F. Migeon, M.-P. Gleizes and F. Gargouri (2015). *ADELFE 3.0 Design, Building Adaptive Multi Agent Systems Based on Simulation a Case Study*. Computational Collective Intelligence: 7th International Conference, ICCCI 2015, Madrid, Spain, September 21-23, 2015, Proceedings, Part I. M. Núñez, T. N. Nguyen, D. Camacho and B. Trawiński. Cham, Springer International Publishing: 19-28.

[8] Winikoff, M. and L. Padgham (2004). *The Prometheus Methodology*. Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. F. Bergenti, M.-P. Gleizes and F. Zambonelli. Boston, MA, Springer US: 217-234.

[9] Giorgini, P., M. Kolp, J. Mylopoulos and M. Pistore (2004). *The Tropos Methodology*. Methodologies and Software Engineering for Agent Systems. F. Bergenti, M.-P. Gleizes and F. Zambonelli, Springer US. 11: 89-106.

[10] Pavón, J. and J. Gómez-Sanz (2003). *Agent Oriented Software Engineering with INGENIAS*. Multi-Agent Systems and Applications III: 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003 Prague, Czech Republic, June 16–18, 2003 Proceedings. V. Mařík, M. Pěchouček and J. Müller. Berlin, Heidelberg, Springer Berlin Heidelberg: 394-403.

[11] Winikoff, M. (2010). Assurance of Agent Systems: *What Role Should Formal Verification Play?* Specification and Verification of Multi-agent Systems. M. Dastani, V. K. Hindriks and C. J.-J. Meyer. Boston, MA, Springer US: 353-383.

[12] Cossentino, M. and C. Potts (2002). PASSI: A process for specifying and implementing multi-agent systems using UML. Retrieved October 8: 2007.

[13] Basin, D., M. Clavel and J. Meseguer (2000). *Rewriting Logic as a Metalogical Framework*. FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science: 20th Conference New Delhi, India, December 13–15, 2000 Proceedings. S. Kapoor and S. Prasad. Berlin, Heidelberg, Springer Berlin Heidelberg: 55-80.

[14] Meseguer, J. (2005). *A Rewriting Logic Sampler. Theoretical Aspects of Computing*. ICTAC 2005: Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005. Proceedings. D. Hung and M. Wirsing. Berlin, Heidelberg, Springer Berlin Heidelberg: 1-28.

[15] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada (2002). *Maude: specification and programming in rewriting logic*. Theoretical Computer Science 285(2): 187-243.

[16] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, José, Meseguer and C. Talcott (2007). *All about maude - a high-performance logical framework: how to specify, program and verify systems in rewriting logic*, Springer-Verlag.

[17] N. Martí-Oliet, José, Meseguer and A. Verdejo (2005). *Towards a Strategy Language for Maude*. Electron. Notes Theor. Comput. Sci. 117: 417-441.

[18] Eker, S., J. Meseguer and A. Sridharanarayanan (2003). *The Maude LTL Model Checker and Its Implementation*. Model Checking Software: 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings. T. Ball and S. K. Rajamani. Berlin, Heidelberg, Springer Berlin Heidelberg: 230-234.

[19] El Fallah-Seghrouchni, A., J. J. Gomez-Sanz and M. P. Singh (2011). *Formal Methods in Agent-Oriented Software Engineering*. Agent-Oriented Software Engineering X: 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers. M.-P. Gleizes and J. J. Gomez-Sanz. Berlin, Heidelberg, Springer Berlin Heidelberg: 213-228.

[20] Ball, E. (2008). An Incremental Process for the Development of Multi-agent Systems in Event-B. Doctoral thesis, University of Southampton.

[21] Ball, E. and M. Butler (2006). *Using Decomposition to Model Multi-agent Interaction Protocols in Event-B*. FM'06 Doctoral Symposium, Springer.

[22] Abrial, J.-R. (2010). *Modelling in Event-B: System and Software Engineering*, Cambridge University Press.

[23] Ball, E. and M. Butler (2009). *Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction*. Methods, Models and Tools for Fault Tolerance. M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna. Berlin, Heidelberg, Springer Berlin Heidelberg: 104-129.

[24] Hadj-Kacem, A., A. Regayeg and M. Jmaiel (2007). *ForMAAD: A formal method for agent-based application design*. Web Intelli. and Agent Sys. 5(4): 435-454.

[25] Graja, Z., A. Regayeg and A. H. Kacem (2011). ForMAAD : *Towards a Model Driven Approach for Agent Based Application Design*. Agent-Oriented Software Engineering XI: 11th International Workshop, AOSE 2010, Toronto, Canada, May 10-11, 2010, Revised Selected Papers. D. Weyns and M.-P. Gleizes. Berlin, Heidelberg, Springer Berlin Heidelberg: 148-164.

[26] Cervenka, R. and I. Trencansky (2007). The Agent Modelling Language - AML: A Comprehensive Approach to Modelling Multi-Agent Systems. Birkhäuser Basel.

[27] Regayeg, A., Hadj-Kacem, A., Jmaiel, M. (2004). *Specification and Verification of Multi-Agent Applications using Temporal Z*. In Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology 2004 (IAT'2004), Beijing, China: 260–266.

[28] Fuxman, A., M. Pistore, J. Mylopoulos and P. Traverso (2001). *Model checking early requirements specifications in Tropos*. Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on, IEEE.

[29] Dardenne, A., A. v. Lamsweerde and S. Fickas (1993). *Goal-directed requirements acquisition*. Selected Papers of the Sixth International Workshop on Software Specification and Design, Elsevier Science Publishers B. V.: 3-50.

[30] Cimatti, A., E. Clarke, F. Giunchiglia and M. Roveri (2000). *NUSMV: a new symbolic model checker*. International Journal on Software Tools for Technology Transfer 2(4): 410-425.

[31] Montali, M., P. Torroni, N. Zannone, P. Mello and V. Bryl (2011). *Engineering and verifying agent-oriented requirements augmented by business constraints with β-Tropos*. Autonomous Agents and Multi-Agent Systems 23(2): 193-223.

[32] Bryl, V., P. Mello, M. Montali, P. Torroni and N. Zannone (2008). *β-Tropos*. Computational Logic in Multi-Agent Systems: 8th International Workshop, CLIMA VIII, Porto, Portugal, September 10-11, 2007. Revised Selected and Invited Papers. F. Sadri and K. Satoh. Berlin, Heidelberg, Springer Berlin Heidelberg: 157-176.

[33] Alberti, M., F. Chesani, M. Gavanelli, E. Lamma, P. Mello and P. Torroni (2008). *Verifiable agent interaction in abductive logic programming: The SCIFF framework*. ACM Trans. Comput. Logic 9(4): 1-43.

[34] Fadil, H. and J.-L. Koning (2005). *A Formal Approach to Model Multiagent Interactions Using the B Formal Method*. Advanced Distributed Systems: 5th International School and Symposium, ISSADS 2005, Guadalajara, Mexico, January 24-28, 2005, Revised Selected Papers. F. F. Ramos, V. Larios Rosillo and H. Unger. Berlin, Heidelberg, Springer Berlin Heidelberg: 516-528.

[35] Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*. Cambridge University Press.

[36] Robinson, K. (1997). *The B method and the B toolkit*. Algebraic Methodology and Software Technology: 6th International Conference, AMAST'97 Sydney, Australia, December13–17, 1997 Proceedings. M. Johnson. Berlin, Heidelberg, Springer Berlin Heidelberg: 576-580.

[37] Jemni Ben Ayed, L. and F. Siala (2008). *Specification and Verification of Multi-agent Systems Interaction Protocols Using a Combination of AUML and Event B*. Interactive Systems. Design, Specification, and Verification: 15th International Workshop, DSV-IS 2008 Kingston, Canada, July 16-18, 2008 Revised Papers. T. C. N. Graham and P. Palanque. Berlin, Heidelberg, Springer Berlin Heidelberg: 102-107.

[38] Bauer, B., J. P. Müller and J. Odell (2001). *Agent UML: A Formalism for Specifying Multiagent Software Systems*. Agent-Oriented Software Engineering: First International Workshop, AOSE 2000 Limerick, Ireland, June 10, 2000 Revised Papers. P. Ciancarini and M. J. Wooldridge. Berlin, Heidelberg, Springer Berlin Heidelberg: 91-103.

[39] Alagar, V. S. and K. Periyasamy (1998). *The Z Notation*. Specification of Software Systems. New York, NY, Springer New York: 281-360.

[40] Regayeg , A., Hadj Kacem ,A., Jmaiel, M, (2005). Towards a formal methodology for developing multi-agent applications using temporal Z. The 3rd ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'05), Cairo, Egypt.

[41] Roungroongsom, C. and D. Pradubsuwun (2015). *Formal Verification of Multi-agent System Based on JADE: A Semi-runtime Approach*. Recent Advances in Information and Communication Technology 2015: Proceedings of the 11th International Conference on Computing and Information Technology (IC2IT). H. Unger, P. Meesad and S. Boonkrong. Cham, Springer International Publishing: 297-306.

[42] Lapouchnian, A. and Y. Lespérance (2009). *Using the ConGolog and CASL Formal Agent Specification Languages for the Analysis, Verification, and Simulation of i* Models*. Conceptual Modelling: Foundations and Applications: Essays in Honor of John Mylopoulos. A. T. Borgida, V. K. Chaudhri, P.

Giorgini and E. S. Yu. Berlin, Heidelberg, Springer Berlin Heidelberg: 483-503.

[43] Yu, E. S.-K. (1996). Modelling strategic relationships for process reengineering. University of Toronto.

[44] De Giacomo, G., Y. Lespérance and H. J. Levesque (2000). *ConGolog, a concurrent programming language based on the situation calculus*. Artificial Intelligence 121(1): 109-169.

[45] Shapiro, S., Y. Lespérance and H. J. Levesque (2002). *The cognitive agents specification language and verification environment for multiagent systems*. Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1. Bologna, Italy, ACM: 19-26.

[46] Wang, X. and Y. Lespérance (2001). *Agent-oriented requirements engineering using ConGolog and i**. Agent-Oriented Information Systems Workshop (AOIS-2001). Montreal, Canada: 59-78.

[47] Lapouchnian, A. and Y. Lespérance (2006). *Modelling Mental States in Agent-Oriented Requirements Engineering*. Advanced Information Systems Engineering: 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006. Proceedings. E. Dubois and K. Pohl. Berlin, Heidelberg, Springer Berlin Heidelberg: 480-494.

[48] Xu, H. and S. M. Shatz (2003). *ADK: An Agent Development Kit Based on a Formal Design Model for Multi-Agent Systems*. Automated Software Engineering 10(4): 337-365.

[49] Deng, Y., S. K. Chang, J. C. A. Figueired and A. Perkusich (1993). *Integrating software engineering methods and Petri nets for the specification and prototyping of complex information systems*. Application and Theory of Petri Nets 1993: 14th International Conference Chicago, Illinois, USA, June 21–25, 1993 Proceedings. M. Ajmone Marsan. Berlin, Heidelberg, Springer Berlin Heidelberg: 206-223.

[50] Stamatopoulou, I., P. Kefalas and M. Gheorghe (2008). *OPERAS: A Framework for the Formal Modelling of Multi-Agent Systems and Its Application to Swarm-Based Systems*. Engineering Societies in the Agents World VIII: 8th International Workshop, ESAW 2007, Athens, Greece, October 22-24, 2007, Revised Selected Papers. A. Artikis, G. M. P. O'Hare, K. Stathis and G. Vouros. Berlin, Heidelberg, Springer Berlin Heidelberg: 158-174.

[51] Eilenberg, S. (1974). *Automata, Languages, and Machines*. Academic Press, Inc.

[52] Bernardini, F. and Gheorghe, M (2004). *Population P Systems*. Journal of Universal Computer Science 10(5): 509–539.

[53] Keller, R. M. (1976). *Formal verification of parallel programs*. Commun. ACM 19(7): 371-384.

[54] Murata, T. (1989). *Petri nets: Properties, analysis and applications*. Proceedings of the IEEE 77(4): 541-580.

[55] Milner, R. (1982). *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc.

[56] Diaconescu, R. and Futatsugi, K. (1998). CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing, vol. 6. World Scientific.

[57] Ölveczky, P. C. and J. Meseguer (2008). *The Real-Time Maude Tool*. Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. C. R. Ramakrishnan and J. Rehof. Berlin, Heidelberg, Springer Berlin Heidelberg: 332-336.

[58] Chella, A., M. Cossentino and L. Sabatucci (2004). *Tools and patterns in designing multi-agent systems with PASSI*. WSEAS Transactions on Communications 3(1): 352-358.

[59] Caire, G., M. Cossentino, A. Negri, A. Poggi and P. Turci (2004). *Multi-agent systems implementation and testing*, na.

[60] Mazouz, M., F. Mokhati and M. Badri (2015). *Towards an Explicit Bidirectional Requirement-to-Code Traceability Meta-model for the PASSI Methodology*. Proceedings of the International Conference on Agents and Artificial (ICAART-2015), Lisbon, Portugal: 203-209.