Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 661–668

ISBN 978-83-60810-22-4

Heterogeneous Formal specification of a Holonic MAS methodology based on Petri Nets and Object-Z

Belhassen MAZIGH
Faculty of sciences,
Department of Computer Science,
5000, Monastir, Tunisia
Email: belhassen.mazigh@gmail.com

Mohamed GAROUI
Faculty of sciences,
Department of Computer Science,
5000, Monastir, Tunisia
Email: garouimohamed2010@gmail.com

Abderrafiaa KOUKAM
University of Technology of
Belfort Montbéliard,
Belfort, France
Email: abder.koukam@utbm.fr

***Abstract*—The objective of this work is to formalize a Holonic methodology called ASPECS by using a formal specification approch based on two formalisms: Petri Net (PN) and Object-Z (OZ) language. Such a specification style facilitates the modeling of complex systems with both structural and behavioral aspects. Our generic approach is illustrated by applying it to FIRA Robot Soccer and it is validated with the SAL framework.**

***Index Terms*—Multi-Agents Systems, Formal Specification, Petri Nets, Object-Z, SAL.**

I. Introduction

IN COMPUTER science, a formal specification is a mathematical description of software or hardware that may be used to develop an implementation. It describes what the system should do but not (necessarily) how the system should proceed. Given such a specification, it is possible to use formal verification techniques to demonstrate that a candidate system design is correct with respect to the specification. This has the advantage that incorrect candidate system designs can be revised before a major investment has been made in actually implementing the design. An alternative approach is to use provably correct refinement steps to transform a specification into a design, and ultimately into an actual implementation that is correct by construction.

A design (or implementation) can never be declared correct in isolation, but only "correct with respect to a given specification." Whether the formal specification correctly describes the problem to be solved is a separate issue. It is also a difficult issue to address, since it ultimately concerns the problem constructing abstracted formal representations of an informal concrete problem domain, and such an abstraction step is not amenable to formal proof. However, it is possible to validate a specification by proving challenging theorems concerning properties that the specification is expected to exhibit. If correct, these theorems reinforce the specifier's understanding of the specification and its relationship with the underlying problem domain. If not, the specification probably needs to be changed to better reflect the understanding domain of those involved in producing (and implementing) the specification.

In specification domain, there are several methodologies to help modely and analyze Multi-Agents and Holonic systems. Among these methodologies, we point out the well known : Tropos [1], PASSI [2] and ASPECS [3, 6].

The objective of this work consists in consolidating an Agent-oriented Software Process for Engineering Complex Systems called ASPECS by using a formal specification and analysis and validate such a specification with the framework SAL (Symbolic Analysis Laboratory).

After a brief presentation of the simulator for the FIRA Robot Soccer competition is presented, a quick overview of the ASPECS process and modelling approach will be presented in section 3. Section 4 present formal specifications of the various domains of ASPECS process based on composition of Petri Nets and Object-Z language named PNOZ. Section 5 present the validation process of PNOZ specification with the framework SAL. Finally, Section 6 summarises the results of the paper and describes some future work directions.

II. Case study: FIRA Robot Soccer

This case study intends to model a simulator for the FIRA Robot Soccer competition [4].

This competition involves two teams of five autonomous robots playing a game similar to soccer (Fig.1). This is a classical example where real-time coordination is required. It constitutes a well-known benchmark for several research fields, such as Multi Agents Systems (MAS), image processing and control.

Robot soccer players are two wheel driven small mobile robots, which are controlled by a host computer. A soccer team in the category of MiroSOT (Micro Robot World Cup Soccer Tournament) consists of five players, one goal keeper and four players for each team. There are several game categories. The size and form of the robot in each category is fixed by rules.

Robot soccer competitions give an opportunity to foster intelligent techniques and intelligent robotics research by providing a standard problem where a wide spectrum of technologies can be developed, tested and integrated, e.g., collaborative multiple agent robotics, autonomous computing,

real-time reasoning and sensor-fusion. From a scientific point of view, the robot soccer players are "intelligent, autonomous agents". They have a global goal like "win the game". Robot soccer has the aim to promote the developments in small autonomous robots and intelligent system (agent) that can cooperate with each other. It belongs to the research topic of MAS.

Fig. 1. FIRA Robot Soccer

III. A QUICK OVERVIEW OF THE USED ASPECS PROCESS

As proposed by Gaud [3] and Cossentino [5], ASPECS is a step-by-step requirement to code software engineering process based on a metamodel, which defines the main concepts for the proposed Holonic Multi-Agents System (HMAS) analysis, design and development based on ***C**apacities*, ***R**ole*, ***I**nteraction* and ***O**rganization* (*CRIO*) framework. The target scope for the proposed approach can be found in complex systems, and especially hierarchical complex ones. The main vocation of ASPECS is towards the development of societies of holonic (as well as not holonic) multiagent systems. ASPECS has been built by adopting the Model Driven Architecture (MDA)[6]. In Cossentino and al. [7] they label the three metamodels "domains" thus maintaining the link with the PASSI metamodels. The three definite fields are:

- *The Problem Domain*. It provides the organizational description of the problem independently of a specific solution. The concepts introduced in this domain are used mainly during the analysis phase and at the beginning of the design phase.
- *The Agency Domain*. It introduces agent-related concepts and provides a description of the holonic, multiagent solution resulting from a refinement of the *Problem Domain* elements.
- *The Solution Domain* is related to the implementation of the solution on a specific platform *Janus* [8, 13]. This domain is thus dependent on a particular implementation and deployment platform.

ASPECS uses UML as a modelling language. Because of the specific needs of agents and holonic organisational design, the UML semantics and notation are used as reference points, which makes ASPECS metamodel ambiguous.

Our contribution will relate to the consolidation of ASPECS process by formalizing the *Problem Domain* and the *Agency Domain* associated to the ASPECS metamodel, therefore facilitating the *Solution Domain*.

IV. FORMAL ASPECS METAMODEL

We use our specification formalism PNOZ introduced in [9, 10] for efficiency modelling and analysis of FIRA Robot Soccer competition. This specification formalism combines two formal languages: Petri Nets (PN) [11] and Object-Z [12].

Our approach consists in giving a syntactic and semantic integration of both languages. Syntactic integration is done by introducing a Behaviour schema based on PN into Object-Z schema. The semantic integration is made by translating both languages towards the same semantic domain : Transition System. To validate our approach, we have limited our work to the specification of the *Team Simulation Organization* which is a part of the holonic structure of the system studied.

A. Petri nets (PN)

Petri nets [16] are 3-tuple $N = (P, T, F)$, where P and T are finite, non-empty, and disjoint sets. P is the set of places and T is the set of transitions. The flow relation between P and T is denoted by $F \subseteq (P \times T) \cap (T \times P)$. The preset of a node $x \in P \cup T$ is defined by ${}^\bullet x = \{y \in P \cup T / (y, x) \in F\}$. The postset of a node $x \in P \cup T$ is defined by $x^\bullet = \{y \in P \cup T / (x, y) \in F\}$. The preset (postset) of a set is defined by the union of the presets (postsets) of their elements. A marking of N is a mapping $M : P \to \mathbb{N}$. (N, M) is called a net system or a marked net. A transition t is said to be enabled if each of its input place p is marked with at least $w(p, t)$ tokens, where $w(p, t)$ is the weight of arc from p to t. $M[t\rangle$ means that transition t is enabled under M. After t fires at M, a new marking M' results. This is denoted as $M[t\rangle M'$. The set of all markings reachable from a marking M_0, in symbols $R(N, M_0)$, is the smallest set in which $M_0 \in R(N, M_0)$ and $M_0 \in R(N, M_0)$ if both $M \in R(N, M_0)$ and $M_0[t\rangle M'$ hold. For a Petri net with n places and m transitions, its incidence matrix A is an $n \times m$ matrix of integers and its typical entry is given by $a_{ij} = a_{ij}^+ - a_{ij}^-$, where $a_{ij}^+ = w(i, j)$ is the weight of arc to place p_i from its input transition t_j and $a_{ij}^- = w(i, j)$ is the weight of arc from place p_i to its output transition t_j. An example of Petri net is shown in Fig.2a and b.

In Fig.2, one can verify that P=$\{p_1, p_2\}$, T=$\{t_1\}$, and F=$\{(p_1, t_1), (t_1, p_1)\}$. Moreover, ${}^\bullet p_1$=$\{t_1\}$, $t_1^\bullet$=$\{p_2\}$, ${}^\bullet p_2$=$\emptyset$, ${}^\bullet p_1$=$\emptyset$, ${}^\bullet t_1$=p_1, ${}^\bullet p_2$=t_1. The initial marking is M_0=$[10]^T$ under which t_1 is enabled since $M(p_1, t_1)$=1 $\geq w(p_1, t_1)$. After t_1 fires, one token is removed from its preceding place, i.e., p_1, and deposited into its succeeding place, i.e., p_2.

B. Object-Z (OZ)

Object-Z [17] is an extension of the Z formal specification language to accommodate object orientation. The essential extension to Z in Object-Z is the class construct, which groups

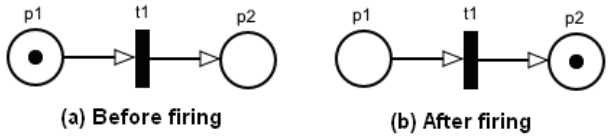

Fig. 2. A Petri net example

the definition of a state schema with the definitions of its associated operations. A class is a template for objects of that class: the states of each object are instances of the state schema of the class, and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class. For example, operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means that no attribute changes value. The standard behavioral interpretation of Object-Z objects is as a transition system [18].

C. Problem Domain associated to our case study

In this section, we use the ASPECS methodology to describe partially the problem domain in the form of *Role*, *Organization*, *Interaction* and *Capacity*. All these concepts are inherited from the classes of *CRIO* framework.

The organization *Team Simulation* (Fig.3) is composed of four roles (*PlayersSimulator*, *RoleAssigner*, *StrategySelector*, *GameObserver*), six interactions between these roles and three capacities (*PlayStrategy*, *ObserveGame*, *ChooseStrategy*) required by the roles.

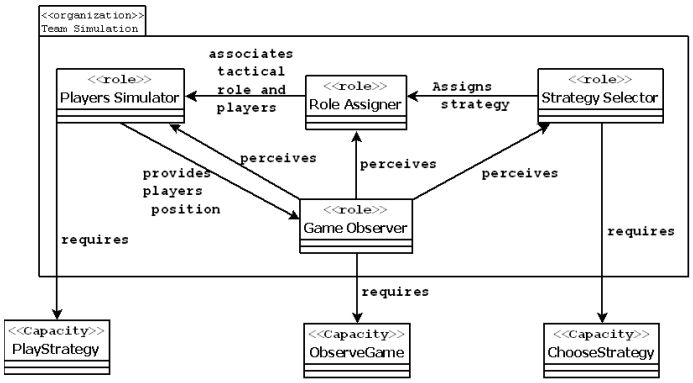

Fig. 3. Team Simulation Organization

Formally, the roles are specified by the Object-Z classes containing behavior schemas. These schemas include the Petri net witch specifies the system's behavior. The incoming and outgoing arrows indicate that the behavior of such a role is related to the behavior of another role. They explain that there is an information exchange between the roles. The class *PlayersSimulatorRole* (Fig.4) specifies the role *PlayersSimulator*. It inherits from the role class and adds these attributes: *requiredCapacity* which is a set of Capacities required by the role.

The role requires a capacity which is named *PlayStrategy*. The behavior of the *PlayerSimulatorRole* specified by the

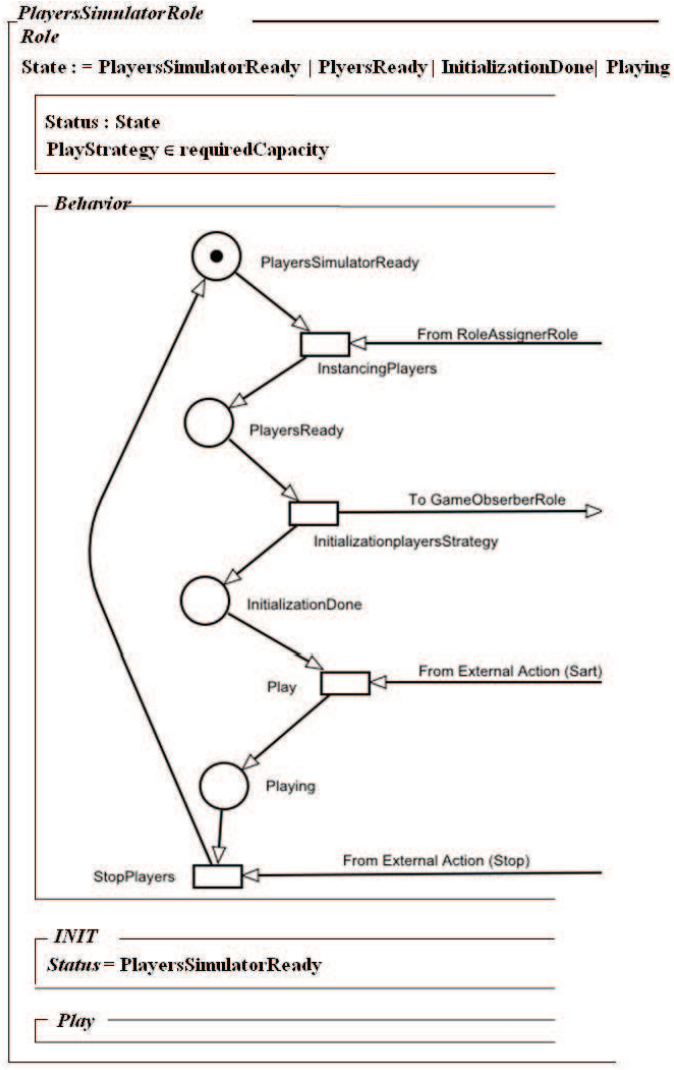

Fig. 4. PlayersSimulatorRole class

behavior schema consists of four states. We suppose that our system can be in these four different states (*PlayersSimulatorReady, PlayersReady, InitializationDone and Playing*). For this reason, we use a type *State* to describe the system states. The incoming and outgoing arrows shows as we said the interconnection between different roles of the same organization. The role *PlayersSimulatorRole* exchange data with the others roles of *Team Simulation* organization such as *RoleAssignerRole*, *GameObserver* Role and other external actions.

The class *PlayStrategy* (Fig.5) specifies the *PlayStrategy Capacity*. Its inputs are a set of *Strategies* and it produces a *PlayingStrategy* as output. The class *Perceives* (Fig.6) specifies the *Perceives Interaction*. It adds the following attributes : *orig* and *dest* to specify the origin and destination of an interaction between two roles.

The other roles of *Team Simulation organization* (Fig.7) will be specified in the same way as *PlayerSimulatorRole*. The class *StrategySelectorRole* (Fig.8), *RoleAssignerRole* (Fig.9)

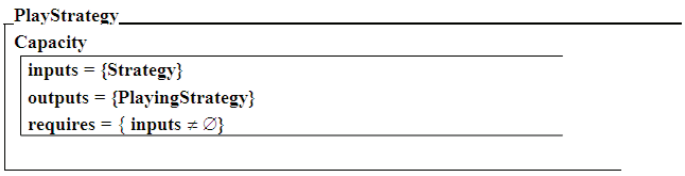

Fig. 5. PlayStrategyCapacity class

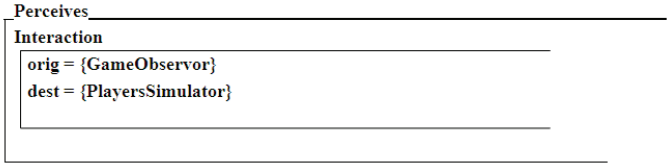

Fig. 6. PerceivesInteraction class

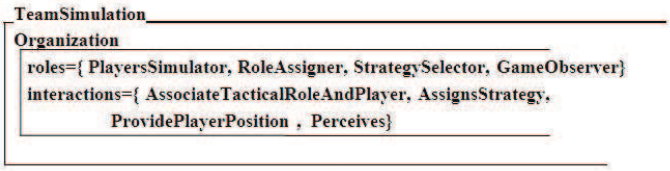

Fig. 7. Team SimulationOrganization class

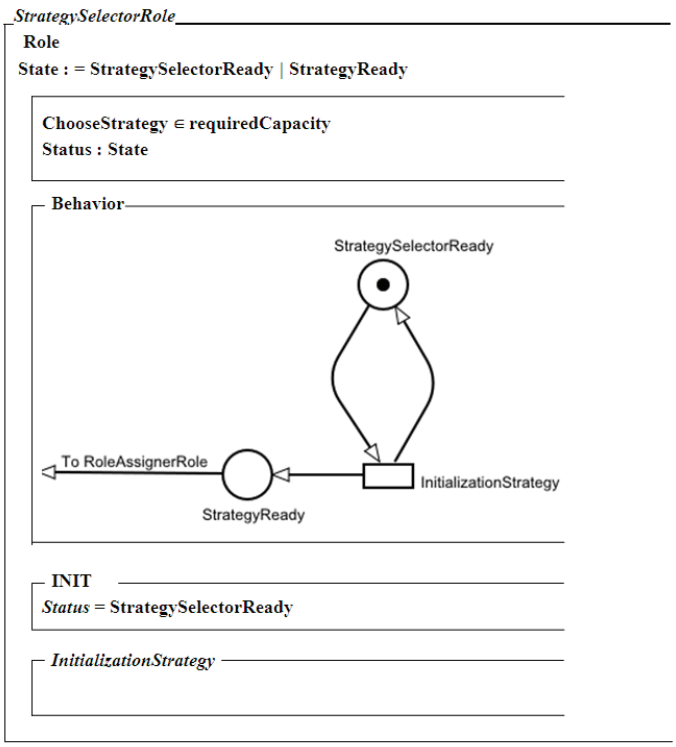

Fig. 8. StrategySelectorRole class

and *GameObserverRole* (Fig.10) specify respectively the role *StrategySelector*, *RoleAssigner* and *GameObserver*.

Remark: The class *TeamSimulation* (Fig.7) specifies the *TeamSimulation* Organization. It has the following attributes:

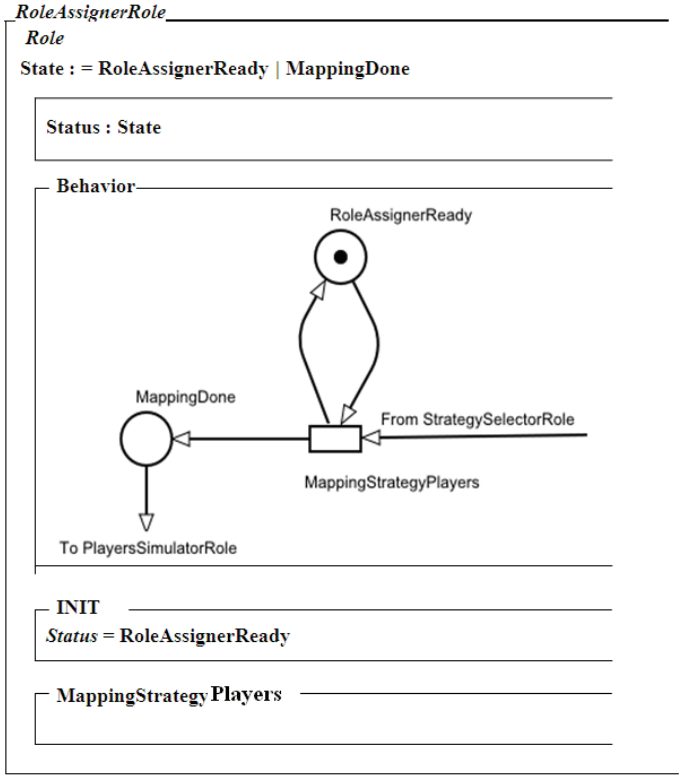

Fig. 9. RoleAssignerRole class

roles representing a set of *Role*, *interactions* representing a set of *Interaction* in the same organization.

D. Agency Domain associated to our case study

In this section, the ASPECS methodology is used to describe partially the analysis phase, the design of the agent society and propose a holonic structure of the FIRA Robot Soccer system.

The Agency Domain includes the elements that are used to define an agent-oriented solution for the problem depicted in the previous stage. Among these elements, we will focus on the most important elements such as *AgentRole*, *Agent*, *Holon*, *HolonicGroup* and *ProductionGroup*.

AgentRole is an instance of the *Problem Domain Role*. It is a behavior and it owns a set of rules in a specific group context.

AutonomousEntity is an abstract rational entity that adopts a decision in order to obtain the satisfaction of one or more of its own goals. An autonomous entity may play a set of Agent Roles within various groups. These roles interact with each other in the specific context provided by the entity itself. The entity context is given by the knowledge, the capacities owned by the entity itself. Roles share this context by the simple fact of being part of the same entity.

An *Agent* is an *autonomousEntity* that has specific individual goals and the intrinsic ability to fulfill some capacities. *Holon* is an *autonomousEntity* that has collective goals (shared by all members) and may be composed of other holons, called

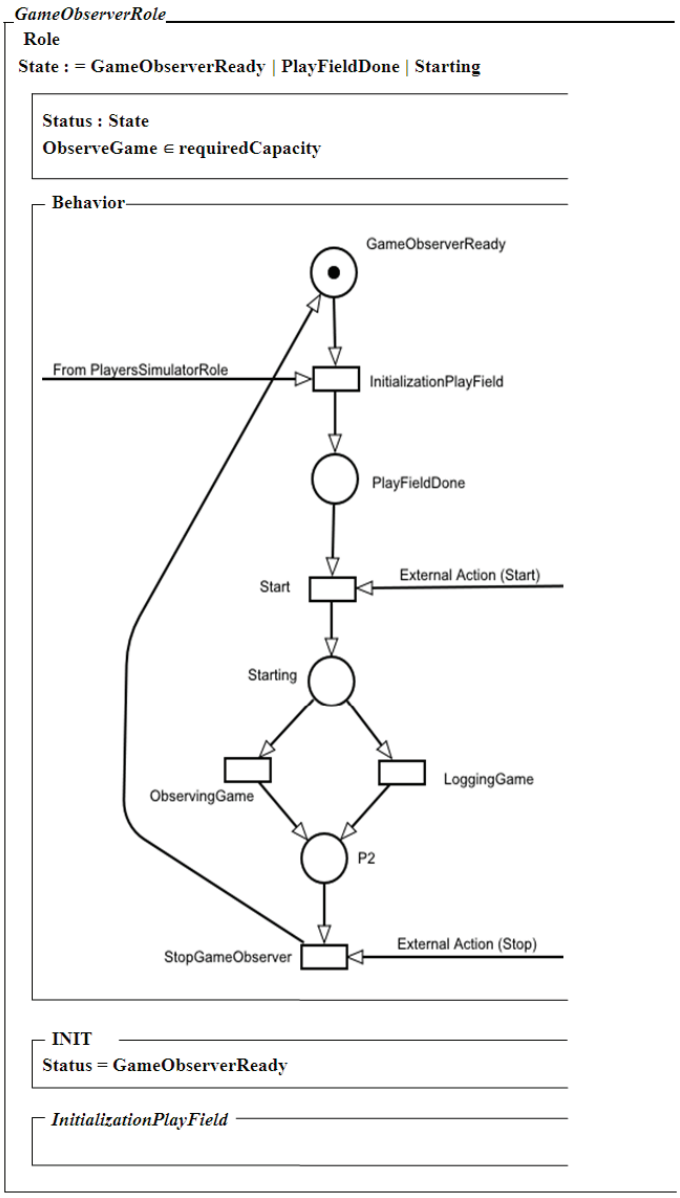

Fig. 10. GameObserverRole class

members or sub-holons. A composed holon is called super-holon. The concept *Group* is an instance in the *Agency Domain* of an *Organization* defined in the *Problem Domain*. It is used to model an aggregation of *AgentRoles* played by *Holons*.

The structure of the holonic solution dealing with our Robot Soccer simulator is presented in (Fig.11), Groups (g1, g5, g3 and g7) are holonic ones (HG). At level 2 of the holarchy, two super-holons H1 and H2 play the role of the *Team* Role in g0 group. Thus, g0 is an instance of a *Game Simulation* Organization. Each of these two super-holons contains an instance of *Team Simulation* Organization (group g2 and g6). Inspired by a monarchic government type, holon members playing the roles of *StrategySelector* (H5 and H9) are automatically named Head and Representative of the other members. Holon Part H3 playing the role of *PlayersSimulator* is decomposed and contains an instance of the *PlayerSimulator* Organization. Its government is inspired by the Apanarchy where all the members are implied in the process of decision-making (all holons are Heads). The atomic holon H6 plays the role of Multipart as it is shared by two couples of super holons (H1, H2) and (H3, H7). This holon represents the environnemental part of the application. The class *PlayersSimulatorAgentRole*

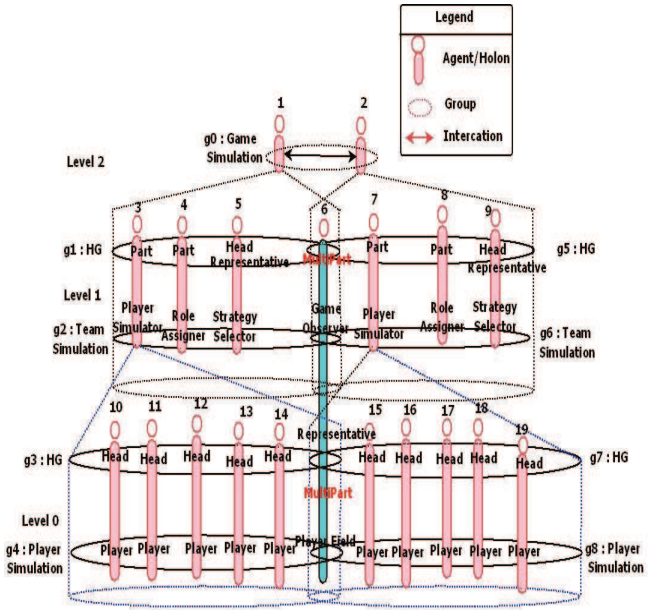

Fig. 11. Holonic Structure of the FIRA Robot Soccer

(Fig.12) specifies the *PlayersSimulator* AgentRole . It has the following attributes: its *requiredCapacity* is a set of Capacities and it provide a services illustrated by a set of services (*providedService*) and *agenttask* is a set of actions specific to agent.

The behavior of the *PlayersSimulatorAgentRole* specified by the behavior schema. This schema contains Petri nets, which are used to describe the behavior of the class *PlayerSimulatorAgentRole*. This last played by the holon H3.

Class H3 (Fig.13) specifies the *PlayersSimulatorHolon* of the holonic structure. It has the following attributes: *agentrole* (a set of played *AgentRole*), *Holonicgroups* (a set of *HolonicGroup*), *productiongroup* (a set of *ProductionGroup*) and collective Goal which cantains all goals of holon.

Class *g3* (Fig.14) specifies the *HolonicGroup g3*. It adds the following attributes: its *members* are a set of *HolonicMembers*.

Class *g4* (Fig.15) specifies the *ProductionGroup g4*. It has the following attributes: its *agents* are a set of *Agent*. Among these agents, class *H11* (Fig.16) specifies *Agent H11*. It introduces the following attributes: its *agentroles* is a set of *AgentRole* played by *Agent H11* and *individualgoal* is a set of *Goals* used by agents to achieve some tasks.

V. Validation of PNOZ Specification with SAL

SAL [14, 15] stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, pro-

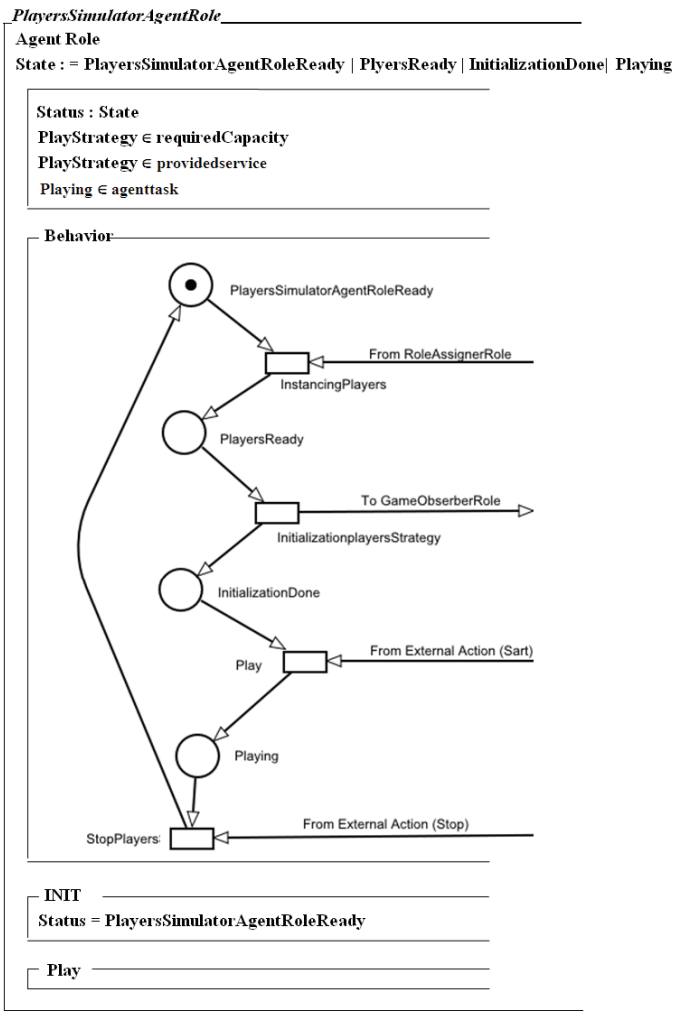

Fig. 12. PlayersSimulatorAgentRole class

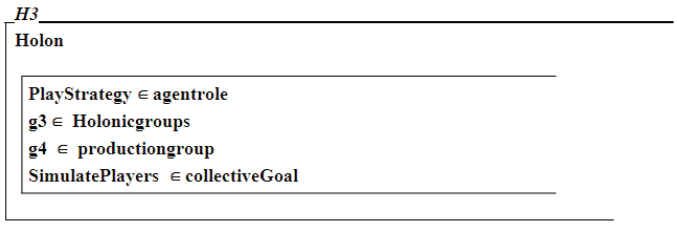

Fig. 13. Holon H3 class

Fig. 14. Holonicgroup g3 class

gram analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition

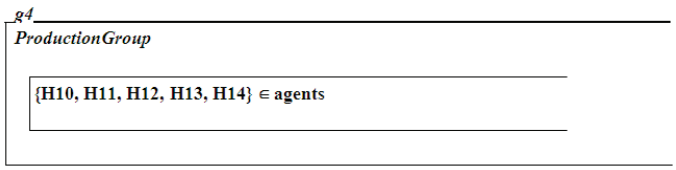

Fig. 15. ProductionGroup g4 class

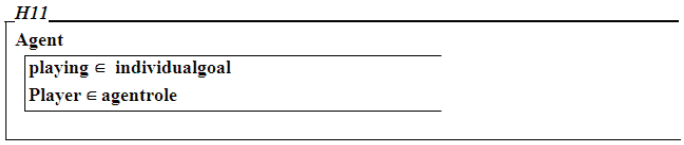

Fig. 16. Agent H11 class

systems. A key part of the SAL framework is a language for describing transition systems. This language serves as a specification language and as the target for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Statecharts. The language also serves as a common source for driving different analysis tools through translators from the SAL language to the input format for the tools, and from the output of these tools back to the SAL language.

The SAL distribution includes the following primary tools : the SAL well-formedness checker (*sal-wfc*), the SAL symbolic model checker (*sal-smc*), sal-deadlock-checker, etc. The SAL well-formedness checker. This is a limited typechecker for SAL. Other SAL tools should not be applied until the errors identified by the well-formedness checker have been corrected. Note that *sal-wfc* is not a full typechecker, so some type-incorrect SAL specifications will escape detection and produce unpredictable results. A BDD, (Binary Decision Diagrams)-based model checker for finite state systems is used. SAL uses the CUDD (CU Decision Diagram) BDD package and provides access to many options for variable ordering, and for clustering and partitioning the transition relation. The model checker can perform both forward and backward search, and also prioritized traversal. *Sal-deadlock-checker*, an auxiliary tool is based on the symbolic model checker for detecting deadlocks in finite state systems. The SAL bounded model checker (*sal-bmc*) is a model checker for finite state systems based on SAT solving. In addition to refutation (i.e., bug detection and counterexample generation), the SAL bounded model checker can perform verification by k-induction. *Sal-path-finder* is an auxiliary tool based on the bounded model checker that generates random paths. The SAL simulator (*sal-sim*) is an interactive front end to other SAL tools.

In this section, we validate our specification with SAL tools. Each specification in PNOZ is transformed into SAL CONTEXT (the framework for declaring types, constants, modules, and module properties).

A. From PNOZ to SAL

This approach of translation allows the flexibility required to directly translate PNOZ specification into SAL. Each OZ class is represented by SAL MODULE (is a self-contained specification of a transition system in SAL) in a specific SAL CONTEXT. The variables of the state schema become local variables of the module, and inputs and outputs of the operations become input and output variables of the module.

Guarded commands may be used in the initialisation and transition sections of a SAL module. The initialisation section of a SAL module may comprise a single guarded command. The transition section may comprise a choice between several guarded commands separated by the syntax "[]". These guarded commands may be labelled to aid the understanding of counter-examples generated by the SAL model checkers.

The part of Petri nets describing the behavior of class PNOZ converted into type represented by an enumerated state of the system and the transitions of Petri net are converted into INPUT variables of type Boolean in state schema of Object-Z class.

Fig.17 represents SAL CONTEXT associated to the class *PlayersSimulatorRole* role. This CONTEXT contains a definition of type "*State*" which is a set that enumerates all possible states of the system. In addition, this CONTEXT also contains a SAL MODULE named *PlayersSimulatorRole* which includes the declaration of Inputs (INPUT), Outputs (OUTPUT) and Locals variables.

Object-Z class operations are converted into a set of transitions in the section TRANSITION of the SAL MODULE. The translation of Object-Z operation schemas into SAL consists of three stages. First, all input and output variables are extracted and converted into their cognate forms in SAL. In addition, each operation schema is converted into a single transition, knowing that the Object-Z schema predicate becomes the guard for the guarded command, expressing the relationship between the primed and unprimed versions of variables. The primed ***status'*** is added to the guard to indicate that the state must hold after firing each transition.

SALenv contains a symbolic model checker called *sal-smc* allows users to specify properties in Linear Temporal Logic (LTL), and Computation Tree Logic (CTL). However, in the current version, SALenv does not print counter examples for CTL properties. When users specify an invalid property in LTL, a counter example is produced. LTL formulas state properties about each linear path induced by a module. Typical LTL operators are:

- $G(p)$ (read "always p"), stating that p is always true.
- $F(p)$ (read "eventually p"), stating that p will be eventually true.
- $U(p, q)$ (read "p until q"), stating that p holds until a state is reached where q holds.
- $X(p)$ (read "next p"), stating that p is true in the next state.

```
PlayersSimulator : CONTEXT =
BEGIN

  State : Type = {PlayersSimulatorReady, PlayersReady,
InitialisationDone, Playing};

      PlayersSimulatorRole : MODULE =

      BEGIN

          INPUT  InstancingPlayers : boolean
          INPUT  InitializationPlayersStrategy : boolean
          INPUT  Play : boolean
          INPUT  StopPlayers : boolean
          OUTPUT status : State

      INITIALIZATION

          status = PlayersSimulatorReady

       TRANSITION [
            Inst_Players : status = PlayersSimulatorReady  and
InstancingPlayers  --> status' = PlayersReady
            []
            Init_Players : status = PlayersReady and
InitializationPlayersStrategy --> status' = InitialisationDone
            []
           Play : status = InitialisationDone and Play  --> status' =
Playing
            []
           Stop : status = Playing and StopPlayers  --> status' =
PlayersSimulatorReady
       ]
       end;

  %------------
  % Properties
  %------------

    livness : theorem system |- G( F(status =PlayersSimulatorReady));
    th1 : theorem system |- G( Play  => F( status = Playing));
    th2 : theorem system |- G(StopPlayers  => F( status =
PlayersSimulatorReady));

end
```

Fig. 17. SAL CONTEXT associated to the class PlayersSimulator

For instance, the formula G(p ⇒ F(q)) states that whenever p holds, q will eventually hold. The formula G(F(p)) states that p often holds infinitely.

B. Properties and proofs

The example illustrated by Fig.17 shows some properties of the system written in the form of theorems with the LTL and CTL formulas. The SAL language includes the clause **theorem** for declaring that a properties is valid with respect to a modeled system by a CONTEXT. These properties can be verified using the following commands:

The first theorem ***th1*** can be interpreted as "whenever Play transition holds, the system will probably in Playing State."The following command line is used:

./sal-smc PlayersSimulator th1
proved.

The second theorem ***th2*** can be interpreted as "whenever StopGame transition holds, the system will probably in Ready State". The following command line is used:

./sal-smc PlayersSimulator th2
proved.

SALenv also contains a Bounded Model Checker called *sal-bmc*. This model checker only supports LTL formulas, and it is basically used for refutation, although it can produce proofs by induction of safty properties. The following command line is used:

./sal-smc PlayersSimulator th2
no counterexample between depths [0, 10].

Remark: The default behavior is to look for counterexample up to depth 10. The option *–depth=<num>* can be used to control the depth of the search. The option *–iterative* forces the model checker to use iterative deepening, and it is useful to find the shortest counterexample for a given property.

Before proving a livness property, we must check if the transition relation is total, that is, if every state has at least one successor. The model checker may produce unsound results when the transition relation is not total. The totality property can be verified using the *sal-deadlock-checker*.The following command line is used:

./sal-deadlock-checker PlayersSimulator PlayersSimulator-Role
Ok (module does NOT contain deadlock state.

The livness theorem can be interpreted as "the initial state of the system is always Ready state". Now, we use *sal-smc* to check the property **livness** with the following command line :

./sal-smc -v 3 PlayersSimulatorRole livness
proved.

C. Compositions and proofs

SAL provides two composition operators for building complex systems from other modules. The asynchronous composition operator is denoted by the syntax "[]". SAL also provides a synchronous composition operator denoted by the syntax "||". The two types of compositions can be freely mixed. For example, one may construct a system as the synchronous composition of two modules that are themselves built by asynchronous composition of other submodules.

The composition operators have the usual semantics. In an asynchronous composition, only one module makes a transition at a time. In a synchronous composition all modules must make simultaneous transitions.

In our work, we will specify *TeamSimulation* Organization with the framework SAL. TeamSimulation Organization is represented with a module SAL named *TeamSimulation*. This module is the main module of specification, which is a synchronous composition of all the modules associated to the roles of this organization: the module *PlayersSimulatorRole* refers to the *PlayersSimulator* Role, the module *StrategySelectorRole* refers to the *StrategySelector* Role, the module *RoleAssignerRole* refers to the *RoleAssigner* Role and the module *GameObserverRole* refers to the *GameObserver* Role.

```
%————————————
%Full system : synchronous composition
%————————————
TeamSimulation : MODULE = PlayersSimulatorRole || StrategySelectorRole || RoleAssignerRole ||GameObserverRole;
```

VI. Conclusion

In this article we showed that HMAS is well adapted to analyse and design the FIRA Robot Soccer competition. The meta-model utilized can be exploited in the implantation stage with the advantage of having formally validated its structure and behaviour by using our composition formalism approach based in Petri Nets and Object-Z named PNOZ. Our future works will focus on a quantative analysis and behavioral validation of different models of ASPECS metamodel. At the same time, it will be interesting to extend Petri Nets with FNLOG (A Logic-Based Function Specification Language).

References

[1] Giorgini, P.: The Tropos Metamodel and its Use, University of Trento, via Sommarive 14, I-38050 Trento-Povo, Italy, 1995.
[2] Azaiez, S.: Approche dirigée par les modeles pour le développement de systèmes multi-agents. PhD thesis, Université de SAVOIE, 1992.
[3] Gaud, N.: Holonic Multi-Agent Systems : From the analysis to the implementation. Metamodel, Methodology and Multilevel simulation. PhD thesis, Université de Technologie de Belfort-Montbéliard, 2007.
[4] FIRA Robot Soccer-ASPECSWiki : www.aspecs.org/FIRA_Robot_Soccer
[5] Cossentino, M., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: A holonic metamodel for agent-oriented analysis and design, 3rd International Conference on Industrial Applications of Holonic and Multi-Agent Systems in LNAI, 4659, Springer-Verlag, 2007.
[6] Object Management Group. MDA guide, v1.0.1, OMG/2003-06-01, 2003.
[7] Cossentino, M., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: ASPECS: an agent-oriented software process for engineering complex systems How to design agent societies under a holonic perspective, Auton Agent Multi-Agent Syst 20:260−304, 2010.
[8] Janus Project : www.janus-project.org
[9] Mazigh, B., Garoui, M.: Formal specification using PN and OZ. Technical Report, Department of Computer Science, Monastir University, 2011.
[10] Mazigh,B., Hilaire, V., Koukam, A.: Formal Specification of Holonic Multi-agent Systems: Application to Distributed Maintenance Company. Published in proceeding of PAAMS 2011, Springer, 2011.
[11] Vinh Duc, T.: Reseau de Petri. Rapport final de TIPE, Institut de la Francophonie pour l'Informatique, 2005.
[12] Duke, R., Rose, G., Smith,G.: Object-Z : A Specification Language Advocated for Description of Standards. Technical report, Software Verification Research Center, Departement of Computer Science, University of Queensland, AUSTRALIA, 1991.
[13] Gaud, N., Hilaire, V., Galland, S., Koukam, A.: An Organizational Platform for Holonic and Multiagent Systems, Multiagent Systems Group, System and Transport Laboratory, University of Technology of Belfort Montbéliard, Published in the Sixth International Workshop on Programming Multi-Agent Systems (ProMAS 08), of the Seventh International Conference on Autonomous agents and Multiagent Systems (AAMAS), 2008.
[14] Natarajan, S.: Symbolic Analysis of Transition Systems, Computer Science Laboratory SRI International, 2000.
[15] Natarajan, S.: Combining Theorem Proving and Model Checking through Symbolic Analysis, Computer Science Laboratory SRI International, Invited Paper at CONCUR 2000, 2000.
[16] Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of IEEE, 77(4), 480−541. 1989.
[17] Duke, R. and Rose, G.: Formal Object Oriented Speci?cation Using Object-Z. Cornerstones of Computing. Macmillan, March 2000.
[18] Smith, G.: A fully abstract semantics of classes for Object-Z. Formal Aspects of Computing, 7(3):289−313, 1995.