BaSi: Multi-Agent Based Simulation for Medieval Battles

Ambra Molesini*, Enrico Denti* and Andrea Omicini†
* ALMA MATER STUDIORUM – Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
Email: {ambra.molesini, enrico.denti}@unibo.it
† ALMA MATER STUDIORUM – Università di Bologna a Cesena
Via Venezia 52, 47521 Cesena, Italy
Email: andrea.omicini@unibo.it

***Abstract*—When dealing with non-trivial social systems, Multi-Agent Based Simulation (MABS) makes it possible to model and simulate social aspects without neglecting articulated motivations, decisions and behaviours by individuals. In this paper, we experiment with the simulation of a peculiar sort of social system – namely, Medieval Battles – by using general-purpose agent methodologies and technologies – namely, SODA and TuCSoN – in order to better understand and emphasise the benefits of MABS in the simulation of social systems.**

I. INTRODUCTION

Simulation aims at modelling and reproducing some natural, ethological, social or conceptual phenomena [1]. The process of designing a simulation usually starts by identifying the features of interest of the system to be modelled, taking into account the desired abstraction level; then, a suitable system representation is defined, the model is built accordingly, and the simulation is finally run based on some carefully-selected input data or application scenarios.

While traditional simulation techniques often model the system dynamics based on systems of equations, this can hardly be made with complex systems, like for instance ecosystems and human communities, where many autonomous individuals interact continuously. In particular, traditional numerical simulations do not consider actions explicitly, nor do they model explicitly interactions among individuals: so, individuals' actions are not perceived as such, but only in terms of impact (the effects) they have on the environment. This prevents the relation between an action and the decisions that determined it (most likely, based on the current environment state) to be suitably expressed. Moreover, numeric simulations are not meant to capture qualitative aspects such as the relation between a stimulus and the consequent behavior, which are often more relevant issues in several complex systems than the quantitative aspect per se.

Multi-Agent Based Simulation (MABS henceforth) promote a different approach to simulation, where interactive entities live, behave and interact in an *agent society* that represents the system to be modelled, making it possible to capture both quantitative and qualitative aspects altogether. In this context, *emerging behaviours* can be observed as the result of individual interactions and choices. The consequent relations emerging between individual behaviour and structural system properties help formulating/validating theories about ethological, sociological, psychological systems.

In this paper, we experiment with MABS by taking *Medieval Battles* as our reference case study. Among the main reasons for our choice:

- it is a non-numeric domain involving both quantitative and qualitative aspects;
- the domain features multiple roles and requires a clear mapping of their relationships;
- the scenario inherently emphasises individual warrior/agent autonomy, while calling for a clear definition of social strategies and tactics;
- there is a widespread focus on interaction issues;
- environment has a prominent role—a particularly interesting issue in MABS;
- the scenario promotes emergent behaviours;
- it is a good testbed from the methodological viewpoint—a relevant aspect indeed, given the amount of related work in the field of agent methodologies for simulation;
- it is a good testbed from the infrastructural viewpoint, as it calls for a powerful and expressive agent coordination platform to actually implement and run the system.

Accordingly, in the remainder of this paper we first (Section II) provide some background about medieval battles in general, and briefly overview the agent methodology (SODA) and infrastructure (TuCSoN) adopted; then (Section III) we focus on the battle simulation system, discussing the whole development process from the requirement analysis to the design, up to the working prototype. Finally, some related work is presented (Section IV), and conclusions are drawn (Section V).

II. BACKGROUND

A. Medieval battles

During the Roman empire, the army was structured according to a rigid, hierarchical organisation, rooted on infantry: discipline, order and organisation provided strong attack and defense force (e.g. the world famous *testudo*). In the subsequent centuries, however, structure and discipline became gradually less relevant, in favor of individual qualities of

soldiers – barbaric armies, indeed, were more like mobs than structured armies. Battles occurred between groups of soldiers, with little or no coordination among them, often without a clear command chain. This aspect became extreme during crusades, which exalted individual heroism. Soldiers were typically armed with swords, halberds, lances, pikes, arches, and crossbows.

The coming of cavalry, from the 5th century, changed the scenario, confining infantry to a complementary role (bowmen and similar "specialised" troops), with the only exception of the siege to a town or castle, where infantry remained obviously essential. Knights were typically equipped with sword, armor, shield, and lance: such a heavy equipment and its maintenance called for both robust horses and auxiliary personnel – a squire, pageboys and servants, all riding on horseback, too – and was all at the knight's expense. This made knights quickly become sorts of "human tanks", virtually impossible to face in an open battlefield: the typical formation consisted of a single knight line, with squires at their back – or, alternatively, at their side. Other times, squires were grouped in small squads, forming a sort of "light cavalry". However, specialised troops – pikemen – constituted a real danger for knights: wisely used, they could lead to the total destruction of the cavalry. In such cases, bowmen troops were used instead of cavalry, saving knights for a later time.

Tactics were quite simple: troops layout were mostly standard, only seldom adapted to the conformation of the ground or other factors, so victory or defeat typically depended on the size of the army and, to some extent, individual qualities – which often turned the battle into a de-structured set of one-to-one duels. As for cavalry, a typical tactic consisted of attacking the enemy and then (falsely) retreating, trying to attract it in pursuit – to counterattack shortly afterwards, exploiting the consequent disorder.

Summing up, due to the lack of discipline, structure and order, even simple tactics could make the difference in medieval battles and often be enough to compensate even quite a large numeric disadvantage. On the other side, however, the lack of coordination in the command chain, coupled with personal visibility goals of individual warriors, could easily vanify the tactic abilities of a commander. So, the final result of a battle was more an emergent behaviour coming out from a collection of individual choices and performance, rather than the expected result of a clearly-planned strategy.

B. The SODA agent-oriented methodology

SODA (Societies in Open and Distributed Agent spaces) [2], [3] is an agent-oriented methodology for the analysis and design of agent-based systems, which adopts the Agents & Artifacts (A&A) meta-model [4], [5] and introduces *layering* as the main tool for scaling with the system complexity, applied throughout the analysis and design process [6].

The SODA abstractions (explained below) are logically divided into three categories: *i)* the abstractions for modelling/designing the system active part (task, role, agent, etc.); *ii)* the abstractions for the reactive part (function, resource, artifact, etc.); and *iii)* the abstractions for interaction and organisational rules (relation, dependency, interaction, rule, etc.). As depicted in Fig. 1, the SODA *process* is organised in two phases, structured in two sub-phases: the *Analysis phase*, which includes the Requirements Analysis and the Analysis steps, and the *Design phase*, including the Architectural Design and the Detailed Design steps. Each sub-phase models (designs) the system exploiting a subset of SODA abstractions: in particular, each subset always includes at least one abstraction for each of the above categories – that is, at least one abstraction for the system active part, one for the reactive part, and another for interaction and organisational rules.

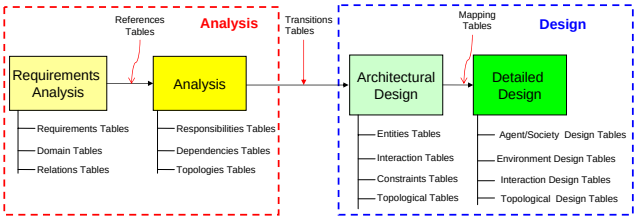

Fig. 1. An overview of the SODA process

C. The TuCSoN agent-oriented infrastructure

TuCSoN (Tuple Centres Spread Over Networks) [7], [8] is an infrastructure for the coordination of distributed autonomous agents via *tuple centres* [9]. Agents access tuple centres associatively, by writing, reading, and consuming tuples via the TuCSoN coordination primitives. A TuCSoN tuple centre is a coordination abstraction perceived by agents as a standard tuple space [10], whose behaviour in response to events can be defined so as to embed the laws of coordination via the ReSpecT language [5].

While TuCSoN features other abstractions and properties that are not of interest here (like the Agent Coordination Context [11]), what is relevant here is the strict relationship between the methodology and the software infrastructure used to design and implement the MABS system [12]: in particular, SODA and TuCSoN share both the conceptual meta-model (A&A [13]) and the focus on the interaction issues.

III. BaSi: Battle Simulation

In this section we first define the intended scenario (Subsection III-A), then proceed from the preliminary problem analysis (Subsection III-B) – independent of the SODA design process – to the problem analysis (Subsection III-C) and the system design (Subsection III-D) – both executed following the SODA process –, up to develop the BaSi prototype (Subsection III-F).

Due to space constraints, only a minimal but meaningful subset of SODA tables and a few screenshots are reported, and only the analysis and design of the simulation sub-system are discussed: the user-interface sub-system is left aside, except for a short description on how the two sub-systems can be integrated.

A. Scenario

Our goal is to realise a simulation system that enables users to *i)* define two armies, *ii)* modify the default properties of the different kinds of soldiers, *iii)* start the battle simulation, *iv)* observe the battle progress and result. Of course, soldiers have to manifest autonomous behaviour, and the simulation has to evolve with no user intervention. The simulation ends when one of the two army is defeated—i.e., all of its soldiers are dead.

Defining an army means to specify at least a) the quantity of soldiers, per type – i.e. the number of knights, bowmen and pikemen, respectively; b) the army commander; and c) the army organisation. With respect to this issue, three different organisations are possible:

- *Informal*: the army does not feature a rigourous organisation: the spatial disposition of soldiers depends on individual decisions, and there is only one general commander in chief.
- *Two-levels*: the army is structured in parties that are typically composed by a uniform kind of soldiers – e.g. knight party, bowmen party and pikemen party. Each party is driven by a party head that refers to the army commander.
- *Three-levels*: the army adopts an even more structured organisation, where parties are split in maniples. Each maniple has its own head, who refers to the party head (who, in turn, refer to the army commander).

Finally, during the battle, soldiers can also pick up abandoned objects, such as equipment from died soldiers (both enemy and friends) like weapons, food, horses, armours, etc.

B. Preliminary analysis

Preliminary requirements analysis suggests that the problem is decomposed in two main sub-problems: *i)* managing the actual simulation (army creation, battle management, etc.); *ii)* managing the user-system interaction (capturing the user commands and graphically rendering the battle). These sub-problems can be assigned to two sub-systems – the *simulation* and the *user-interface* sub-systems, respectively – to be designed independently from each other, while taking into proper account the information flow between them.

Quite expectedly, all the core aspects of this paper are in the first sub-system: in fact, the user-interface sub-system (despite its potential graphical complexity) is simply concerned with getting initial parameters from the user and providing graphical results, and therefore does not require autonomous/intelligent entities; its design can then follow a standard object-oriented process, and will not be discussed here.

The simulation sub-system, instead, is well suited for an agent-oriented approach, as the agent abstractions seem particularly adequate to capture the key aspects of medieval battles:

- agents' autonomous and intelligent behaviour can easily model the soldiers: in turn, this makes it easy to model the army informal organisation, where each agent is able to decide its disposition in the battlefield and its actions during the fight;
- agent societies can well capture the social aspects of the army, such as its social goals (e.g. enemy destruction) and the social rules that govern its organisation: in particular, agent societies can map also the cases where the army organisation changes during the battle, thanks to agent's adaptability;
- the multiagent system (MAS henceforth) environment can capture both the topological aspect of the battlefield and the presence of the abandoned objects, which covers the last key aspect of the simulation sub-system.

As a preliminary step of the design process of the simulation sub-system, a soldier model, intended as a collection of his relevant properties, has to be defined. For our purposes, we assume henceforth the following soldiers' characterisation:

- *Type*: the soldier type (knight, bowman or pikeman)
- *Army*: the belonging army
- *Speed*: the maximum soldier speed in the battlefield
- *Vigour*: the maximum tolerable damage (before dying)
- *Attack strength*: the maximum damage inflicted to an enemy
- *Defence strength*: the maximum decrease of the damage caused by an enemy
- *View scope*: the maximum distance at which the soldier can see
- *Attack scope*: the maximum distance at which the soldier can attack
- *Killing*: the number of enemies killed when attacking
- *Loading*: the maximum load that the soldier can carry
- *Equipment*: the list of the soldier's objects (weapons, food, money, etc.)

Each equipment object corresponds to a bonus/malus increment in some soldiers' properties: for instance, the presence of a horse increases the soldier's speed and view scope, which is instead decreased by a heavy armour; food increases the soldier's vigour; etc. Of course, each property (including bonus/malus coefficients) has a default value, that can be modified by the user before starting the simulation.

C. Analysis

Analysis in SODA consists of two sub-phases: *Requirements Analysis* and *Analysis*.

Requirements Analysis. Following the choices discussed in Subsection III-B, here we focus on the simulation sub-system, as the user-interface sub-system can be considered a part of the system environment, wrapped by a suitable artifact: so, its presence appears only in terms of its interactions with the other SODA entities in the design process. Its observable behaviour will then constitute one of the requirements of its own (object-oriented) design process, which is not discussed here.

The simulation system requirements at the core layer are reported in Fig. 2. This is intentionally a high-level view, aimed at highlighting the main coarse-grained requirements: so, just three items – army, simulation, and monitoring – are listed.

Such requirements will then be refined later in the process, exploiting SODA *layering* – i.e., its ability to support different levels of detail.

Several relations exist among such requirements: for instance, there is clearly an order relation between "Army Definition" and "Simulation", as between "Simulation" and "Monitoring". Detecting such relations at this stage is important, as they will impose constraints and interactions in the following steps.

Requirement	Description
Army Definition	definition of the armies in terms of soldiers and their parameters
Simulation	management of the simulation
Monitoring	monitoring the battle progress and identification of the army winner

Fig. 2. Requirement table.

Analysis. In this phase, the above requirements are mapped onto *tasks*, and are further analysed to identify the *environmental functions* and the *topological structure of the environment*; dependencies among tasks and functions are also individuated.

Tasks identified in this phase are usually more fine-grained than requirements: yet, they are still at a relatively high abstraction level, to be in-zoomed later in the process. Fig. 3 reports the mapping between the requirements and the tasks they generate in our case.

Requirement	Task
Army Definition	define_knight, define_bowman, define_pickman, define_party define_maniple, define_head, attack, defence, collaboration
Simulation	start, stop, pause
Monitoring	show_status, check_soldier, check_progress

Fig. 3. Reference Requirement-Task table.

Environmental functions can be classified here basically in two categories: *i)* functions belonging to the internal MAS environment (battlefield management, soldiers' property management, etc. – Fig. 4 top), and *ii)* functions provided by the user-interface sub-system (Fig. 4 bottom).

Dependencies among tasks and functions exist, and must be carefully investigated as they determine the interaction spaces of each entity, the rules that govern interactions, and the related social aspects (like army organisation management). In our case, for instance, soldiers' movements strictly depend on the orders issued by the army's (or party's, or maniple's) commander; moreover, even the fights could be modelled as a dependency involving soldiers of the enemy armies. Further dependencies are generated by the functions of the user-interface sub-system, and represent the information exchanged between the simulation and the user: some occur in the user $\rightarrow$ simulation direction (update the soldier's parameters, starting/stopping the simulation, etc.), others in the opposite one (everything representing the battlefield state to be graphically rendered: soldier's position and energy, position of abandoned objects, etc.).

Function	Description
State Area	providing the state of a specific battlefield area
Update Soldier State	updating soldier state
Battlefield State	providing battlefield state
Update Battlefield State	updating battlefield state
Rendering	showing the battlefield state changes
Command	obtaining user command
Ext Update Soldierd	obtaining user modifications to soldier parameters

Fig. 4. Function table.

Topological aspects are also extremely relevant in our scenario, since soldiers can engage a fight only if they are close enough to each other, and the battle strategy itself depends on (and is strictly related to) the topology of the enemy army. So, in this application the role of the environment topology is twofold: on the one hand, it determines the "physical" structure of the battlefield, in terms of the "zone" that can be perceived by each soldier; on the other, it determines the constraints over the soldiers' actions and perceptions in terms of the soldiers' "scopes". Such scopes can be composed by multiple zones, depending on each soldier's characteristics and equipment: for instance, a knight can perceive larger areas thanks to his higher position, while bowmen's attacks can go farther, etc.

D. Design

Design in SODA consists of two sub-phases, too: *Architectural Design* and *Detailed Design*.

Architectural Design. In this phase, the system is designed in terms of *roles*, *resources*, *actions*, *operations*, *interactions*, *rules* and *spaces*. These entities derive form the abstractions outlined in the previous step: roles are responsible for the achievement of tasks by executing actions, resources provide the functions by implementing the corresponding operations, spaces derive from the topology and map one-to-one the physical structure of the battlefield. Mappings between tasks and roles, and between functions and resources, instead, are not usually one-to-one, as a role/resource is able to complete/accomplish several different tasks/functions: for instance, the resource "Battlefield" could provide both the "Battlefield State" and "Update Battlefield State" functions in Fig. 4.

The key aspect of the system – interaction – is captured by the SODA abstract entities *interactions* (derived from dependencies), and *rules* (derived from both dependencies and topologies): again, such mappings are expressed by suitable tables. Fig. 5 shows an excerpt of the full *Rule table*, reporting some representative rules for each "macro area": the first block concerns the battle rules (Attack Rule and Win Rule), the second is about the internal management of an army (PickUp Rule and Army Head Rule), the third concerns the order relations highlighted in the Requirements Analysis phase (Start Rule and Monitoring Battlefield Rule).

Rule	Description
Attack Rule	The attack action is possible only if the enemy is in the scope of the attacker
Win Rule	An Army wins the battle only if the number the enemy army soldiers reaches zero
PickUp Rule	A soldier can pick up an abandoned object if this last is near to the soldier
Army Head Rule	The army head's command is the more priority command
Start Rule	The simulation starts only if the state of Interface Resource is start
Monitoring Battlefield Rule	The state of the BattleField Resource can change only if the simulation is started

Fig. 5. Rule table.

Detailed Design. In this phase, one Detailed Design has to be chosen from the various potential alternatives compatible with the Architectural Design constraints. Detailed Design is expressed in terms of *agents*, *agent societies*, *artifacts*, *aggregates* and *workspace* for the architectural entities, and of concepts such as *use*, *manifest*, *speakTo* and *linkedTo* for interaction types. So, moving from the Architectural Design to the Detailed Design means to decide a mapping for all the architectural entities and abstractions onto actual design entities, deciding which level of detail is the most adequate for each one. The activity of selecting the most adequate representation level for each architectural entity is called *carving*.

In our case, however, since a very high abstraction level was adopted for the core layer, and no in-zoom operation was performed, carving is quite straightforward: most of the architectural design entities can be mapped 1-1 onto corresponding Detailed Design entities. So, agents in BaSi play simply the roles individuated in the previous step, while resources are mapped onto suitable *environmental artifacts*—a kind of artifact [14] which is particularly suited for wrapping MAS external environments, or realising functions derived by requirements. (In fact, an environmental artifact is perfect for wrapping the user-interface sub-system as a black-box, according to the choices made in the Requirements Analysis.) Spaces and space connections are also mapped 1-1 onto workspaces and workspace connections.

The only real carving decision concerns Agent Societies: taking inspiration from Fig. 3, we opted for just two agent societies — one "Army Society" grouping all the agents performing army-related tasks (first line of Fig. 3), and one "Simulation Society" grouping all the agents performing management-related tasks (second and third lines of Fig. 3). So, "Army Society" is composed of agents representing the army commanders, parties, maniples, knights, bowmen, and pikemen, while "Simulation Society" includes all the other agents responsible for the simulation management.

The mapping of interactions takes into account the different nature of interaction acts: so, agent-agent interactions are mapped onto *speakTo* entities, agent-artifact interactions onto *use* entities, artifact-agent interactions onto *manifest* entities, and artifact-artifact interactions onto *linkedTo* entities.

Rules, too, are mapped taking into account the different nature and purpose of each rule. In particular, rules aimed at controlling the interaction of a single agent within the MAS are mapped onto the agent's *individual artifact* [14]—a kind of artifact associated to each agent and used as a "proxy" between the agent and the MAS, to shape and negotiate the set of admissible agent actions in/onto the MAS; in BaSi, this is the case, for instance, of the "Attack Rule" and the "PickUp Rule" (Fig. 5). Instead, rules concerning social and organisational aspects (such as "Win Rule" and "Army Head Rule" in Fig. 5) are mapped onto *social artifacts* [14]—a kind of artifact specifically devoted to the management of social interactions[1] which is typically used in SODA as a coordination medium, encapsulating the laws that govern an agent society. In BaSi, the two agent societies defined above ("Army Society" and "Simulation Society") require two social artifacts: we call these "Army Artifact" and "Simulation Artifact", respectively. As a further design choice inspired by a conceptual economy principle, we decide that these artifacts also take care of the two above-mentioned organisational rules ("Win Rule" and "Army Head Rule"), which actually concern the same sets of agents; of course, different choices would be possible, with pros and cons.

Despite all this complexity, the design presented here is just a partial view of the actual system (Subsection III-F): in the full scenario, agents in each army could be organised according to specific military structures, which could not be reasonably managed by a single social artifact for performance and reliability reasons—preventing bottlenecks, avoiding delays in the system responses, etc. So, the "Army Artifact" should rather be seen as an abstract view of an *aggregate* of social artifacts, each enforcing some given kind of rules.

Fig. 6 shows an excerpt of the Artifact-UsageInterface table, which lists all the operations provided by all artifacts in detail.

Artifact	Usage Interface
Army Artifact *(social)*	receive_headCommand send_headCommand add_soldier, remove_soldier set_strategy, get_strategy get_position, set_position...
Simulation Artifact *(social)*	start_simulation, stop_simulation, pause_simulation...
Properties Artifact	set_property, get_property add_soldierType, add_property get_soldierTypes, get_armyTypes...

Fig. 6. Artifact-UsageInterface table.

[1]This often occurs indirectly, since social artifacts technically mediate interactions between individual, environmental, and possibly other social artifacts.

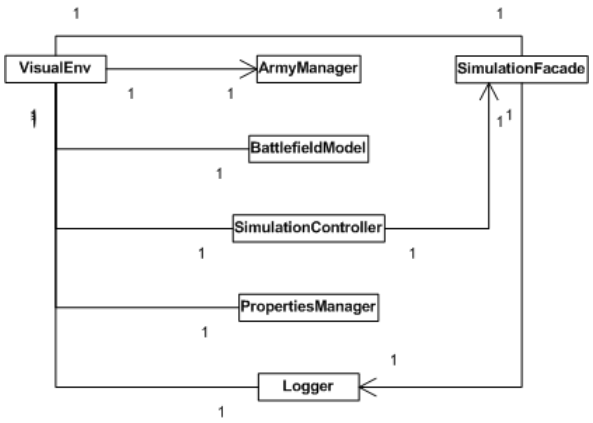

Fig. 7. An overview of the user-interface sub-system

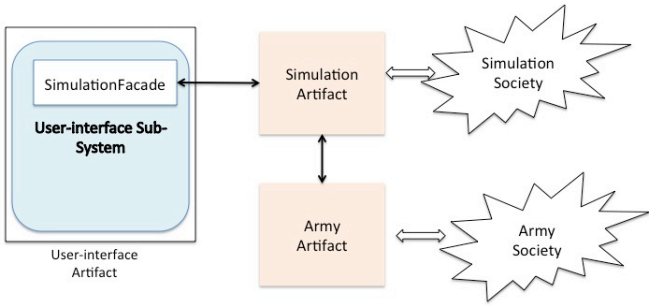

Fig. 8. An overview of the composition structure

E. Sub-systems composition

According to the general choices made in the Preliminary Analysis, the BaSi development has been split in one agent-oriented process for the simulation sub-system (discussed above), and one more "classical" object-oriented process for the user-interface sub-system. Although a full discussion of the latter is outside the scope of the paper, its outcome is necessary to integrate the two sub-systems together. As shown in Fig. 7, the user-interface sub-system is made of seven components, each responsible for a given functionality: *VisualEnv* is the graphical renderer that animates and updates the graphical elements in the battlefield, *PropertiesManager* enables the modification of the simulation parameters, *ArmyManager* provides for army creation, *BattlefieldModel* stores the battlefield data, *SimulationFaçade* takes care of the data exchange with the simulation sub-system, *SimulationController* handles the user commands to activate, pause and deactivate the simulation, and finally *Logger* logs all the activities of all components.

The consequent integration, presented in Fig. 8, is straightforward, since the user-interface sub-system behaviour and its interactions with the simulation sub-system were both carefully designed during the SODA process: qualitatively speaking, the user-interface sub-system is wrapped by an ad-hoc *User-interface Artifact*, which puts that sub-system inside the MAS. Interactions take care of the information exchange among the two sub-systems: in particular, *linkedTo* interactions connect *SimulationFaçade* to the *Simulation Artifact*, so that the user commands coming from the GUI components (*SimulationController* or *ArmyManager*) are properly forwarded to *Simulation Artifact* via *SimulationFaçade*, and hance dispatched to the Simulation Society and Army Society, as required — and vice versa.

F. Prototype

Given the goals of our work, focuses on MAS technologies for simulation, only a minimal effort was devoted to the graphical rendering: as shown in Fig. 9, the battlefield and soldiers are shown with simple icons – squares represent knights, circles represent bowmen, triangles represent pikemen –, and so are other aspects, like heads and commanders (which are rendered as edged icons, like the knight of the Red Army in Fig. 9 – a), soldiers' energy level (rendered as black lines over the icons, in percentage terms), and the soldier's current energy level (number inside the icon).

Before the simulation can start, the user has to specify some initial parameters – army names, armies' organisational structures (to be chosen among *informal*, *party* or *maniple*), and battlefield size – and can optionally change the default values of soldier properties (Fig. 9 - c). If the informal army organisation has been selected, only one commander can be indicated; otherwise, party heads and maniple heads can also be specified for each party or maniple, respectively.

The simulation can be started, paused and stopped using the appropriate controls (Fig. 9 - b)): at any time, the battle progress can be monitored. The simulation automatically stops when an army wins — that is, all of its soldiers are dead.

Technically, the prototype is structured according to the UML diagram in Fig. 10, and is implemented on top of the TuCSoN infrastructure, whose tuple centres are used to build the social artifacts and support/mediate between the information exchange. The diagram highlights the information exchange between the two sub-systems, and the key role played by TuCSoN tuple centre for this purpose and for realising the social artifacts, managing the agents coordination and enforcing the organisational rules via the ReSpecT reactions.

IV. Related Work

A. Simulation frameworks for social systems

A specific area of agent-oriented computing where simulation finds many applications is that of social systems, where specific simulation frameworks have been employed.

Sierra et al. present an integrated development environment for the engineering of MASs as Electronic Institutions (EI) [15], [16]. This includes SIMDEI, a simulation tool which allows for the animation and analysis of the specification of the rules and protocols in an EI. In this approach, once specified an institution, it should go through a verification process to verify it. After an initial verification process focusing on static, structural properties of the e-institution specification, the next step follows that concerns the expected dynamic properties of the e-institution at work: this is done by means of simulation, with the aim of verifying the dynamic properties of

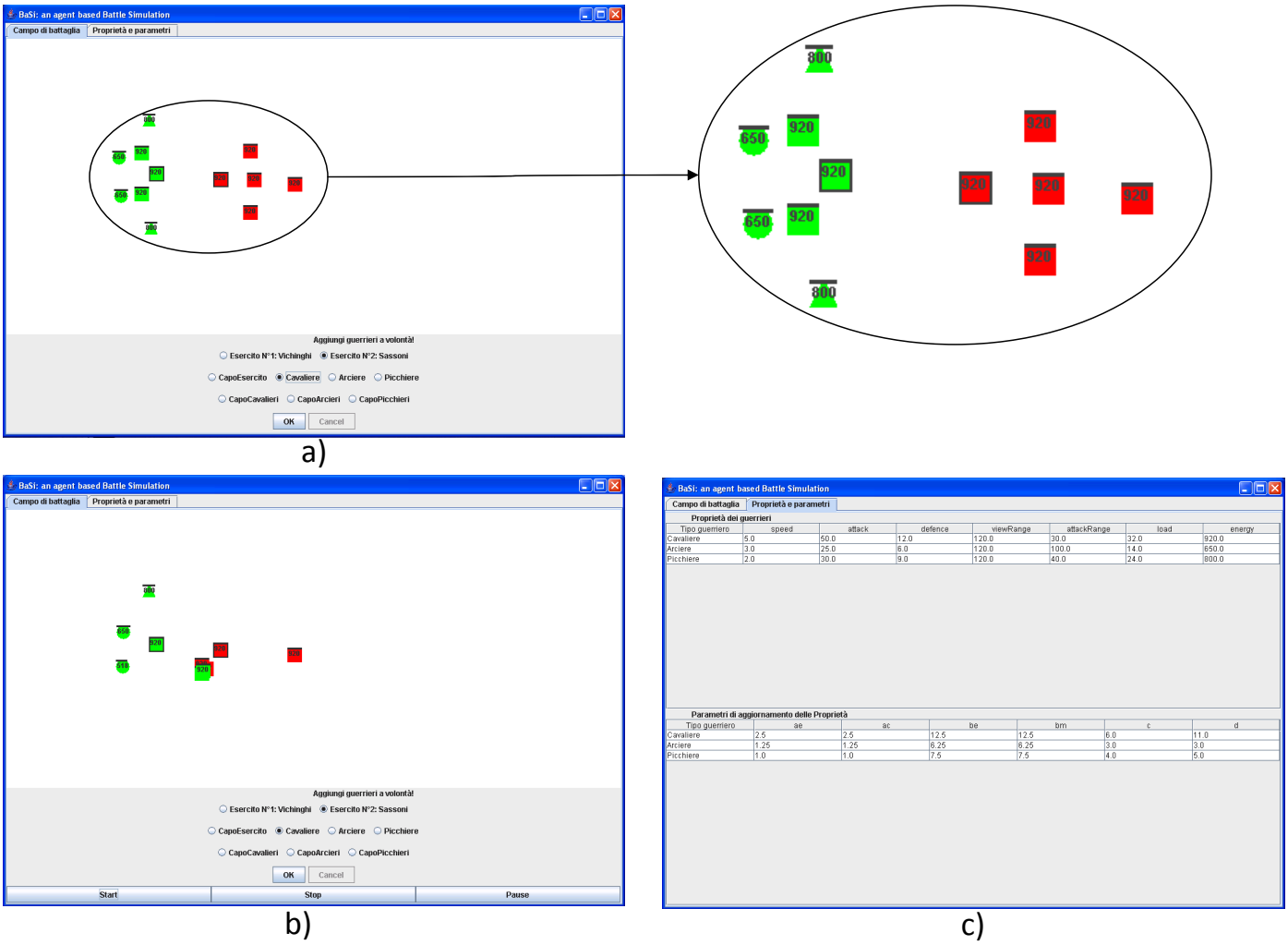

Fig. 9. Some screenshot of BaSi: a) army deployment interface, b) main interface during simulation, c) tuning parameters and property interface

the specified institution. Once agents have been implemented, simulations of the e-institution can be ran using the SIMDEI simulation tool developed over Repast (Recursive Porous Agent Simulation Toolkit). The institution designer should analyse the results of the simulation and return to initial step, if they differ from the expected ones.

In Pavòn et al. [17], a simulation phase based on the agent-based simulation toolkit Repast is defined and introduced for the INGENIAS [18] methodology for the development of MASs. The main objective is to support modelling and simulation of social systems. In particular the authors modified the INGENIAS MAS meta-model in many ways: *i)* environment model, since for social simulation, agents usually require to consider their location in the environment and the evolution of time; *ii)* modelling constant time steps to simulate the cycle perception-reaction of agents along the time, since authors have assumed time driven simulations as a reference. In addition, the authors have created a mapping from the new INGENIAS to the Repast toolkit, implemented by an IDK module.

In Röhl and Uhrmacher [19] a modelling and simulation framework (DynDEVS) based on a discrete-event formalism for supporting the development process of multi-agent systems from specification to implementation is proposed. The framework allows for the incremental refinement of agents and experimental set-ups while providing rigorous observation facilities. The exploited simulation framework is JAMES, a Java-Based Agent Modelling Environment for Discrete Event Systems Specification (DEVS)-based Simulation, which aims at exploring the integration of the agents paradigm within a general modelling and simulation formalism for discrete-event systems. Devs (Discrete EVent System specification) is one of the formal approaches to discrete event modelling and simulation stemming from general systems theory. It provides a powerful basis for modelling test settings by being able to encode many other modelling formalisms like statecharts and petri nets.

Sarjoughian et al. [20] presents a layered architectural framework to support agent-based system development in a collaborative, multidisciplinary engineering setting: the en-

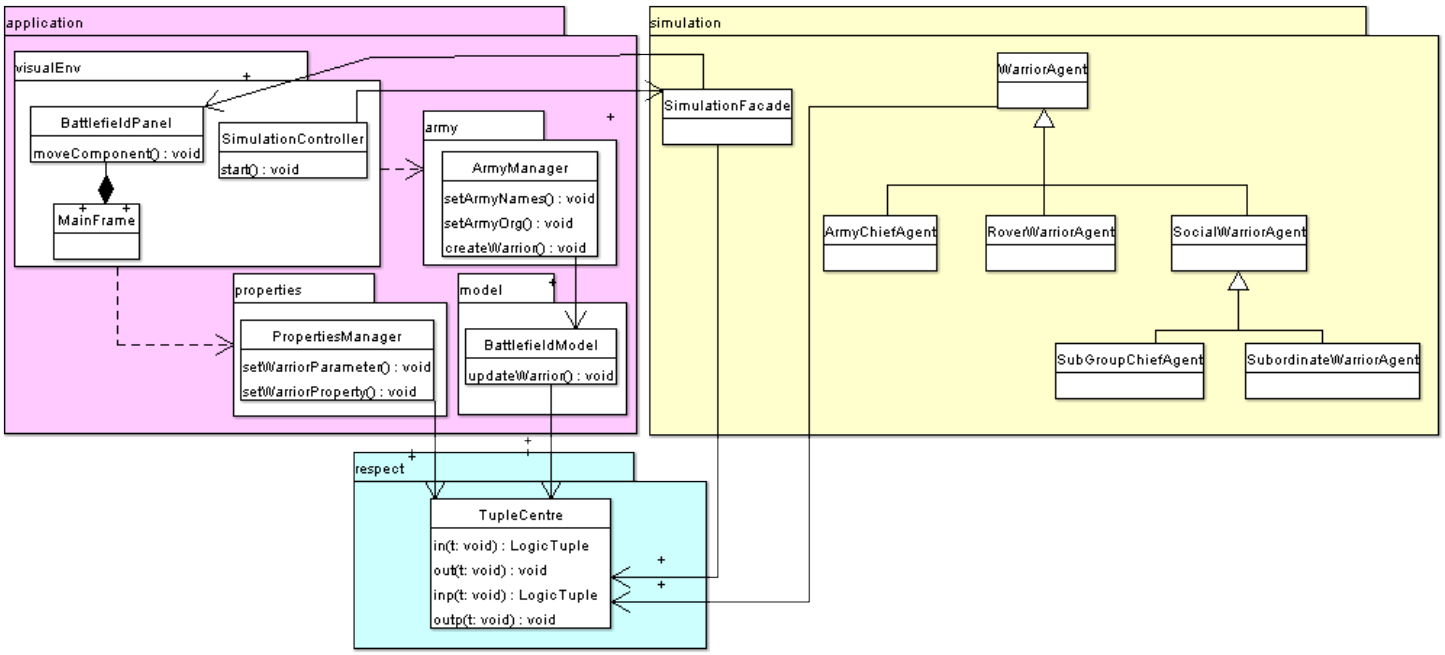

Fig. 10. An overview of the prototype structure

vironment is assumed to enable agent-based modelling and simulation Authors consider requirements for a generic, comprehensive modelling and simulation architecture suitable for MAS, and emphasise the need for separating modelling and simulation activities, which is argued to have a profound impact on reusability and portability.

B. Simulation in agent methodologies

Some proposals exist whose goal is to provide general AOSE methodologies incorporating a simulation phase.

In Fortino et al. [21], [22] an integrated approach is presented, centred on the instantiation of a software development process which specifically includes a simulation phase used to validate a multi-agent system before its actual deployment and execution. The approach uses process fragments coming from the Gaia [23] methodology for the analysis and the design, the Agent UML and the Distilled StateCharts for the detailed design, the MAO Framework for the neutral- platform implementation of software agents, and a Java-based discrete-event simulation framework for the simulation. In particular, the simulation phase provides both qualitative and quantitative information about correctness and efficiency, to be exploited for reformulating, modifying and/or refining some choices of the previous phases of the development process. An agent-based system is validated and evaluated by implementing a simulator program, whose execution provides a history of the timed events generated and received by the agents; the analysis of the trace file can be used to validate the correctness of the agent interactions and behaviours.

The work in Cossentino et al. [24] proposes the Process for Agent Specification, Simulation and Implementation (PASSIM), a simulation-based process for the development of MASs which incorporates a simulation phase for the prototyping of the MAS being developed and for functional and nonfunctional validation. PASSIM was obtained by integrating – according to a process-driven method engineering approach – fragments coming from two existing agent-oriented methodologies: on the one hand, the PASSI methodology [25] carries out the analysis, design and coding phases, while the above-mentioned Distilled State Charts (DSC)-based simulation method [21], [22] is used for supporting the simulation phase. PASSIM is supported by MASSIMO (Multi-Agent System SIMulator framewOrk) [22], a Java-based discrete-event simulation framework for MASs which allows for the validation and evaluation of: the dynamic behaviour of individual and cooperating agents; the basic mechanisms of the distributed architectures supporting agents, namely agent platforms; the functionalities and emergent behaviours of applications and systems based on agents. The simulation results can be used to feed back the Simulation Model Definition.

easyABMS [26] is another agent-based methodology for the modelling and simulation of complex systems, which seamlessly covers from the system analysis to the system modelling phase, up to the analysis of the simulation results. Each phase of the easyABMS (iterative) model-driven process refines the model produced in the previous phase. While its work-products are mainly visual (UML) diagrams, the simulation code is also automatically generated, thanks to the advanced features of visual modelling and of (semi)automatic code generation provided by the Repast Simphony Toolkit.

Summing up, the SODA approach turns out to be quite different from the three above methodologies. In fact, SODA does not consider simulation as a specific part of the process design, although we are currently working on an extension for

introducing a specific simulation phase: so, the outcomes of the SODA process are not currently validated by a simulation phase as they are in the other cases. Moreover, unlike Repast, the TuCSoN infrastructure is not specifically designed for simulation, either: so, our work can be seen as a first experiment to explore how SODA and TuCSoN can support the development of a MABS system, aimed at a better understanding of the SODA limits in the simulation scenario and plan its future extension.

V. Conclusions and Future Work

In this paper we present a first application prototype for the simulation of medieval battles. The main objective of this work was to investigate the suitability of MABS in the development of an articulated scenario such as medieval battles. So, we deliberately left apart aspects that are normally relevant for a simulation system, such as a rigourous and efficient simulation engine, the realisation of both a complex graphical interface and a detailed animation of the graphical elements.

Yet, the experiment turned out to be an interesting testbed for the SODA methodology, highlighting some benefits as well as some limitations that we plan to address in the near feature. In particular, the tabular representation is clearly more suitable for an automatic tool than for a human designer, due to the large amount of tables to be filled in at each stage: so, tools will be developed that support designers in this task in a consistent and complete way. Another interesting extension could be the definition – or the adoption – of a language for specifying SODA rules and interactions in a more precise and formal way, overcoming the implicit limitations of the natural language which is currently adopted in the tabular representation. We also plan to evaluate whether to enrich SODA with methods for the internal design of agents and artifacts—which are now uncovered, since SODA currently does not deal with intra-agent (and more generally with "internal") issues.

With respect to the simulation context discussed in this paper, further work will be devoted to improve the prototype, improving the simulation engine and adopting a better graphical rendering engine. More complex and intelligent behaviours for soldiers could also be added, as well as user functionalities for defining specific military strategies for each army.

Acknowledgements

Authors would like to thank Dr. Alberto Mercati for his contribution to the project and his work on the prototype implementation.

References

[1] A. Drogoul and J. Ferber, "Multi-agent simulation as a tool for modeling societies: Application to social differentiation in ant colonies," in *Selected papers from the 4th European Workshop on on Modelling Autonomous Agents in a Multi-Agent World, Artificial Social Systems*. London, UK: Springer-Verlag, 1994, pp. 3–23. [Online]. Available: http://portal.acm.org/citation.cfm?id=646907.710638

[2] A. Omicini, "SODA: Societies and infrastructures in the analysis and design of agent-based systems," in *Agent-Oriented Software Engineering*, ser. LNCS, P. Ciancarini and M. J. Wooldridge, Eds. Springer, 2001, vol. 1957, pp. 185–193, 1st International Workshop (AOSE 2000), Limerick, Ireland, 10 Jun. 2000. Revised Papers.

[3] SODA, "Home page," `http://soda.apice.unibo.it`. [Online]. Available: http://soda.apice.unibo.it

[4] A. Molesini, A. Omicini, E. Denti, and A. Ricci, "SODA: A roadmap to artefacts," in *Engineering Societies in the Agents World VI*, ser. LNAI, O. Dikenelli, M.-P. Gleizes, and A. Ricci, Eds. Springer, Jun. 2006, vol. 3963, pp. 49–62, 6th International Workshop (ESAW 2005), Kuşadası, Aydın, Turkey, 26–28 Oct. 2005. Revised, Selected & Invited Papers. [Online]. Available: http://www.springerlink.com/link.asp?id=j68l84713542525p

[5] A. Omicini, "Formal ReSpecT in the A&A perspective," *Electronic Notes in Theoretical Computer Sciences*, vol. 175, no. 2, pp. 97–117, Jun. 2007, 5th Inter. Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 Aug. 2006. Post-proceedings.

[6] A. Molesini, A. Omicini, A. Ricci, and E. Denti, "Zooming multi-agent systems," in *Agent-Oriented Software Engineering VI*, ser. LNCS, J. P. Müller and F. Zambonelli, Eds. Springer, 2006, vol. 3950, pp. 81–93, 6th International Workshop (AOSE 2005), Utrecht, The Netherlands, 25–26 Jul. 2005. Revised and Invited Papers. [Online]. Available: http://www.springerlink.com/link.asp?id=h6529n587642uh24

[7] A. Omicini and F. Zambonelli, "Coordination for Internet application development," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, pp. 251–269, Sep. 1999.

[8] "TuCSoN home page," `http://tucson.apice.unibo.it`.

[9] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, January 1985.

[10] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992.

[11] A. Omicini, "Towards a notion of agent coordination context," in *Process Coordination and Ubiquitous Computing*, D. C. Marinescu and C. Lee, Eds. Boca Raton, FL, USA: CRC Press, Oct. 2002, ch. 12, pp. 187–200.

[12] A. Molesini, E. Denti, and A. Omicini, "From AO methodologies to MAS infrastructures: The SODA case study," in *Engineering Societies in the Agents World VIII*, ser. LNCS, A. Artikis, G. O'Hare, K. Stathis, and G. Vouros, Eds. Springer, Sep. 2008, vol. 4995, pp. 300–317, 8th International Workshop (ESAW'07), 22–24 Oct. 2007, Athens, Greece. Revised Selected Papers. [Online]. Available: http://www.springerlink.com/content/yh7x3253p4j535r9/

[13] A. Omicini, A. Ricci, and M. Viroli, "Artifacts in the A&A meta-model for multi-agent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 17, no. 3, pp. 432–456, Dec. 2008, special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems. [Online]. Available: http://www.springerlink.com/content/l2051h377k2plk07/

[14] ——, "*Agens Faber*: Toward a theory of artefacts for MAS," *ENTCSs*, vol. 150, no. 3, pp. 21–36, 29 May 2006.

[15] C. Sierra, J. A. Rodríguez-Aguilar, P. Noriega, M. Esteva, and J. L. Arcos, "Engineering multi-agent systems as electronic institutions," *European Journal for the Informatics Professional*, vol. 4, 2004.

[16] J. L. Arcos, J. A. Rodríguez-Aguilar, and B. Rosell, "Engineering autonomic electronic institutions," in *Engineering Environment-Mediated Multi-Agent Systems*, ser. LNCS, D. Weyns, S. A. Brueckner, and Y. Demazeau, Eds., vol. 5049. Springer, 2008, pp. 76–87.

[17] J. Pavòn, C. Sansores, and J. J. Gòmez-Sanz, "Modelling and simulation of social systems with ingenias," *Int. J. Agent-Oriented Softw. Eng.*, vol. 2, no. 2, pp. 196–221, 2008.

[18] J. Pavòn, J. J. Gòmez-Sanz, and R. Fuentes, "The INGENIAS methodology and tools," in *Agent Oriented Methodologies*, B. Henderson-Sellers and P. Giorgini, Eds. Hershey, PA, USA: Idea Group Publishing, Jun. 2005, ch. IX, pp. 236–276. [Online]. Available: http://www.idea-group.com/books/details.asp?id=4931

[19] M. Röhl and A. Uhrmacher, "Controlled experimentation with agents - models and implementations," in *Engineering Societies in the Agents World V*, ser. LNCS, M. P. Gleizes, A. Omicini, and F. Zambonelli, Eds., vol. 3451. Springer, 2005, pp. 292–304.

[20] H. Sarjoughian, B. Zeigler, and S. Hall, "A layered modeling and sim-

ulation architecture for agent-based system development," *Proceedings of the IEEE*, vol. 89, no. 2, pp. 201–213, Feb 2001.

[21] G. Fortino, A. Garro, and W. Russo, "From modeling to simulation of multi-agent systems: An integrated approach and a case study," in *Multi-agent System Technologies*, ser. LNCS, G. Lindemann, J. Denzinger, I. J. Timm, and R. Unland, Eds., vol. 3187. Springer, 2004, pp. 213–227.

[22] ——, "An integrated approach for the development and validation of multi-agent systems," *Comput. Syst. Sci. Eng.*, vol. 20, no. 4, 2005.

[23] F. Zambonelli, N. Jennings, and M. Wooldridge, "Multiagent systems as computational organizations: the Gaia methodology," in *Agent Oriented Methodologies*, B. Henderson-Sellers and P. Giorgini, Eds. Hershey, PA, USA: Idea Group Publishing, Jun. 2005, ch. VI, pp. 136–171. [Online]. Available: http://www.idea-group.com/books/details.asp?id=4931

[24] M. Cossentino, G. Fortino, A. Garro, S. Mascillaro, and W. Russo, "Passim: a simulation-based process for the development of multi-agent systems," *International Journal of Agent-Oriented Software Engineering*, vol. 2, no. 2, pp. 132–170, 2008.

[25] M. Cossentino, "From requirements to code with the PASSI methodology," in *Agent Oriented Methodologies*, B. Henderson-Sellers and P. Giorgini, Eds. Hershey, PA, USA: Idea Group Publishing, Jun. 2005, ch. IV, pp. 79–106. [Online]. Available: http://www.idea-group.com/books/details.asp?id=4931

[26] A. Garro and W. Russo, "easyabms: A domain-expert oriented methodology for agent-based modeling and simulation," *Simulation Modelling Practice and Theory*, vol. 18, no. 10, pp. 1453–1467, 2010, simulation-based Design and Evaluation of Multi-Agent Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X10000717

[27] B. Henderson-Sellers and P. Giorgini, Eds., *Agent Oriented Methodologies*. Hershey, PA, USA: Idea Group Publishing, Jun. 2005. [Online]. Available: http://www.idea-group.com/books/details.asp?id=4931