Designing More Reliable MAS-based Ambient Intelligence Systems

Juan Carlos AUGUSTO a,1, Miguel J. HORNOS b,
a C.S.R.I. and School of Computing and Mathematics,
University of Ulster, UK
b Software Engineering Department, University of Granada, Spain

Abstract. We explore the availability of tools which can support the development of more correct and reliable Multi-Agent Systems (MAS) immersed in Ambient Intelligence (AmI) systems. We explain how state of the art software engineering methods and tools can be used to guide the development of such systems and increase our confidence on them. The explanation assumes little technical background on behalf of the reader. We use this exercise as an opportunity to highlight the advantages of available tools as well as to illustrate why the AmI area should build a new generation of development frameworks.

Keywords. Reliability, Multi-Agent Systems (MAS), Ambient Intelligence (AmI)

Introduction

Ambient Intelligence (AmI) Systems are starting to populate our society [1,2]. These systems can be defined as “A digital environment that supports people in their daily lives by assisting them in a sensible way” [3]. Given their intimate relation with humans these systems should be considered as safety-critical.

The software industry has developed tools in the past few decades to help developers examine the quality of software. Companies use this software to aid the development of products, especially those which, given their characteristics are especially complex. Real-time and distributed systems have provided problems where this technology has thrived [4].

This chapter will explore the value of current state of the art technology which can support development of AmI systems, in particular we will show how the increasingly popular multi-agent based development approach to develop such systems can be also guided by such software engineering paradigms and supporting tools.

We assume a reader interested in this book will have some knowledge on Multi-Agent Systems (MAS) but not necessarily on the software engineering methods and tools which are used to explain the development techniques. The MAS system we use as a case study is related to the development of an Ambient Assisted Living project. As such we believe this provides an ideal context given the need to increase reliability in the area [5].

1E-mail: jc.augusto@ulster.ac.uk.
First we describe the relationship between MAS and AmI, then we explain methods and tools which can be used to support development of MAS systems for AmI and finally we illustrate with an example how this idea can be deployed.

1. Different models of MAS for AmI

There are good reasons why developers and researchers interested in AmI systems use multi-agent systems to build their software applications. Software agents have characteristics (e.g. distributed, reactive, capable of learning, social awareness) which are compatible with the general requirements of AmI systems. In the area of software for smart homes for example, we can observe three main types of organization or strategies being followed to organize the software agents: sensor centred, room centred, and task centred.

The MavHome architecture [6] uses a mix of levels: physical (sensors and actuators), communication, information, and decision agents. Our case study will be based on [7] so we will explore the task centred approach in the rest of this chapter. This makes the results less dependent on the technology and the specific house where it is used.

2. Software reliability: Methods and tools

Software Engineering has provided Computer Science with important methods and tools to increase reliability in software and computing systems. The reader can find a good summary on these advances in [4,8,9]. These tools can be used to model some aspects of an AmI system and make its behaviour a bit more predictable. Methods and tools for verification are usually applied following a standard procedure, see Figure 1 for a global view of the typical process that is usually followed (this is fairly standard and independent of the verification technique and tool used).

First a system is specified or modelled in some notation with a clear syntax and semantics, often with the help of a tool. Then the behavioral properties that are relevant in
that system are expressed in a language (usually different from the one used to characterize the system) which will also have precise syntax and semantics. Then a tool helps to analyze if the relevant behavioral properties are present in the specified system or not. As a result of that analysis the tool may be able to either prove that the property holds with regards to the system specified or to provide a counterexample showing that is not the case.

The existing verification frameworks present differences among them, mainly on the adopted languages for modelling, simulation and verification. Typical modelling notations are along the lines of automata or procedural representations of the processes involved. Simulation facilities are provided in a variety of ways according to the type of systems the verification framework can deal with. Verification facilities are usually tied to some sort of implicit or (more often) explicit temporal logic language. The type of temporal concepts which can be handled by this language dictates the type of properties a user can aim to check by using that specific verification framework. We have chosen SPIN as the verification framework to exemplify our proposal, the next section gives more details on how modelling, simulation and verification are used in SPIN.

3. SPIN

SPIN [10] is one of the most well-known and used systems for software verification. It is a highly efficient and stable system with a user friendly interface, and good team support. The tool is freely available and only relies on having pre-installed some C++ related files (e.g. available installing MinGW), a self installing interface based on Tcl/Tk (also freely available and self-extracting). SPIN is focused on the concept of models. Users can model a system by using a language called PROMELA (acronym for PROcess MEta-LAnguage), which emphasizes the role of processes and their interactions. Once a user has built a model, the possible scenarios represented in the model can be simulated in various ways (e.g. randomly guided by the machine or user guided). Additionally users can perform what in Software Engineering is called formal verification, i.e. an exhaustive analysis of all the possible executions implied by the model. SPIN provides a formal language for users to specify properties which can then be checked by the tool. If the property the user was trying to verify in the model does not hold then SPIN provides a counterexample (i.e. an example which explains why that property is not true for that particular model).

3.1. Modelling

As we have mentioned above, SPIN users must model the systems to be verified (and/or simulated) using its modelling language, called PROMELA [11,10]. Although its syntax is C-like, it is not an implementation language, but a specification or description language which is intended to easily make good abstractions of the system to be modelled. As its main target is the design of concurrent software systems (like the Nocturnal MAS-based system shown in Appendix A), PROMELA allows adequately modelling the coordination and synchronization of processes. However, it is not focused on computation, reason why this language has few computational functions and lacks floating point numbers.

To model a system with PROMELA (like the one shown in Appendix A), software designers must use three elements: processes, message channels, and variables. While
processes are intended to specify a given behaviour, message channels and variables are
used by processes as communication and coordination means, and define the environ-
ment in which the processes run.

All the processes are global objects representing the concurrent entities of a dis-
tributed system. The behaviour of each process is defined by a proctype declaration
(see "Process Declaration Section" in Appendix A). When the active keyword pre-
fixes a process definition, an instance of that process will be active in the initial state.
Observe that all the processes in Appendix A have been defined in this way.

Message channels and variables can be declared either globally (see the two first sec-
tions in Appendix A) or locally within a process (see e.g. the declaration of the Boolean
variable activeC in the process Client). Different basic data types (bit, bool, byte, mtype, short, and int) can be used for declaring variables, though all those
appearing in Appendix A have been defined as Boolean variables (using the keyword
bool). Likewise, although message channels can be defined to be either synchronous
(i.e. unbuffered, written a 0 in brackets) or asynchronous (i.e. buffered, written a number
greater than 0 in brackets), all the channels defined in Appendix A are synchronous. This
allows implementing a rendezvous port between processes.

Message channels are used to transfer or pass data (in first-in-first-out order) be-
tween processes and to synchronize them, using the sending (!) and receiving (?)
operators. Thus, e.g. the statement tia2client?intervention (see proctype
Client definition in Appendix A) retrieves a message from the head of the chan-
nel tia2client, and stores it in the variable intervention, while the statement
client2sensors!event sends the value of the expression event to the channel
client2sensors, i.e. it appends such value to the tail of this channel. Process in-
teractions via synchronous channels will do that sender or receiver (the one that arrives
first at the channel) will be blocked, waiting for the counterpart (receiver or sender) that
arrives second.

Two control flow constructs have been used in the model of Appendix A: the
case selection (if ... fi) and the repetition or loop (do ... od). The selec-
tion structure must contain at least two execution sequences, each preceded by a dou-
ble colon (: :). Only one sequence from the list will be non-deterministically selected
and executed. A sequence can be selected only if its first statement, which is called a
guard, is executable. For example, in the expression !activeC -> activeC=true,
activeC is the guard and must be true (i.e. activeC must be false) before the state-
ment activeC=true can be executed. If all guards of a selection structure are unexe-
cutable, the process will block until one of them can be selected. The dummy statement
skip (used in some of the selection structures in Appendix A) is always executable and
has no effect.

The repetition structure is in some sense an extension of the selection structure, since
only one option can be selected at a time. After this option is completely executed, the
execution of the structure is repeated. The usual way to finish the execution of the rep-
etition structure is with a break statement, which transfers the control to the instruc-
tion that immediately follows the repetition structure. Observe that all the processes de-
defined in Appendix A are always running, since there is not any break statement in the
repetition structure which defines the behaviour of each process.

\[2\] Here \(!\) represents the negation operator.
Users can indicate that a sequence of statements has to be executed as one indivisible unit (i.e. non-interleaved with any other processes) by enclosing in curly braces the sequence and prefixing it with the keyword `atomic`. Doing this, if any statement in an atomic sequence (except the first one) blocks, a runtime error will occur. Atomic sequences can be used for reducing the complexity of verification models, since they restrict the amount of interleaving that is allowed in the system.

3.2. Simulation

A PROMELA model describing the possible behaviour of a concurrent system is executable in some sense, such that the simulator included in SPIN can provide the user some insight on the system reactions on the basis of certain scenarios. These scenarios can be either randomly generated or provided by the user, as we will see just below.

SPIN can carry out three types of simulations:

- **Random simulation**, which by default makes use of the current time to seed the random number generator that SPIN uses to produce a new simulation run. As a result, every new simulation run may generate a different execution of the model. However, the user can also define the seed value to make the simulation run reproducible.

- **Interactive simulation**, where a menu with several choices is offered to the user at each point of the execution where the simulator can proceed in more than one way. Thus, all non-deterministic choices in the model which were automatically resolved in the previous mode calling to the random number generator, here are determined by the user’s direct selection.

- **Guided simulation**, which requires a trail file to guide the simulation. This file is generated only in verification mode, when the verifier of a model finds a violation of the property to be checked, storing the execution sequence leading up to the error in the file. This sequence therefore constitutes a counterexample that the user can visualize using this simulation mode.

To conclude this subsection we must say that simulation is adequate for a quick first assessment of the quality of a model, but it is less useful to find subtle errors, since it is generally infeasible (and even impossible) to simulate all the possible trails of a model. Consequently, simulation can only help to debug the models, but no amount of simulation runs can prove the properties we may want to. If this is our goal, we have to apply verification.

3.3. Verification

This section is centred on the process of predicting the behaviour of a system based on a model. The model is studied rigorously to discover logical flows in the strategy that represents and in the assumption that if the real system is implemented following that idea faithfully then the implementation in the real world will have the same properties which have been found in the model.

The great contribution of this scientific community in the last few decades is that tools such as SPIN [10] have been developed based on algorithms [12,13,14] linking automata theory, which can be used as a formal representation of the model, with a
logical description of the property to be checked [15]. Fundamental advances in this area have been recognized recently with significant awards to pioneers in the area: [16], [17], and [18].

To specify the properties to be verified SPIN users must use Linear Temporal Logic (LTL) [15,19], which supplements Propositional Logic\(^3\) with temporal operators for the future (e.g. $\Diamond$, $\Box$, $\bigcirc$ and $U$). Here we need to clarify that these operators are adapted in SPIN so that they can be expressed in ASCII notation; as a result, they are respectively represented as $<>$, $[]$, $X$ and $U$. Likewise, the classical logical symbols are also adapted to this notation; thus, $\land$ becomes $\&$, $\lor$ becomes $\mid\mid$, $\rightarrow$ becomes $-\rightarrow$, and $\neg$ becomes $!$.

The mentioned temporal operators are read as follows (the operator name is written in italics):

- $<>p$ “eventually in the future $p$ holds”
- $[]p$ “always in the future $p$ holds”
- $Xp$ “at the next state $p$ holds”
- $pUq$ “$p$ is true until $q$ is true”

The informal semantics of these temporal operators is as indicated in Figure 2. $\bigcirc q$ (i.e. $Xq$ in SPIN notation) is fulfilled at the state $n-3$, since $q$ holds at the next state ($n-2$). The formula $\Diamond r$ (or $<>r$) is satisfied if and only if there is at least a state in the future (including the current state) where $r$ holds; this formula is therefore fulfilled at states from 0 to $n$, since $r$ holds at state $n$. The formula $\Box p$ (or $[]p$) is satisfied at a state if $p$ permanently holds from that state; thus, since $p$ remains true forever from state $n-1$, this formula meets in all these states (observe that it will be satisfied in state $n-2$ only if $p$ also holds in that state). Finally, $sUq$ means that $s$ has to hold from the current state until the one where $q$ holds (this can occur at the current or a future state); consequently, this formula is fulfilled at the three first states shown in the figure.

\[\begin{align*}
&\Box p \quad p \quad p \quad p \quad p \quad p \quad \ldots \\
&\bigcirc q \quad q \\
&\Diamond r \quad r \\
&sUq \quad s \quad s \\
\end{align*}\]

![Figure 2. Informal interpretation of temporal operators](image)

The formal semantics of the temporal operators presented is given by:

\[\begin{align*}
\models f &\iff s_0 |\models f \\
\models Xf &\iff s_{i+1} |\models f \\
\models [f &\iff \forall j (j>=i) : s_j |\models f \\
\models <>f &\iff \exists j (j>=i) : s_j |\models f \\
\models eUf &\iff \exists j (j>=i) : s_j |\models f \text{ and } \forall k (i<=k<j) : s_k |\models e
\end{align*}\]

\(^3\)We will assume the reader has a basic understanding of logic.

\(^4\)Where $s$ is a sequence of states ($s_i$) and $|=\text{ represents the satisfiability relation.}
The value of this theory relies on its capacity to provide a specialized language to describe situations which can then be accurately checked. After extensive use in the community of Software Engineering, some formulas of this language have been found particularly useful and of widespread use, see e.g. [20]. Some of those patterns of behaviour, which are especially interesting and which can be represented in the logical language adopted by SPIN, are:

- **Safety:** \([\neg p]\)
- **Guarantee:** \(\langle\langle p \rightarrow \neg q\rangle\rangle\)
- **Response:** \([\langle\langle p \rightarrow \neg q\rangle\rangle]\)
- **Persistence:** \(\langle\langle\neg p\rangle\rangle\)
- **Recurrence:** \(\langle\langle\neg p\rangle\rangle\)

Examples for the three more common patterns in the context of a Smart home can be:

- “Is it true that the front door cannot be open and closed at the same time in this model?”, i.e. \([\neg (\text{frontDoorOpen} \land \text{frontDoorClosed})]\)
- “Will the fridge door be open at some future instant?”, i.e. \(\langle\langle \text{open_fridge} \rangle\rangle\)
- “Whenever the smoke alarm is triggered the system sends an SMS through the mobile phone”, i.e. \([\langle\langle \text{smokeAlarmTriggered} \rightarrow \text{sendSMS_smokeAlert} \rangle\rangle]\)

Notice here we are not considering hard temporal constraints, this is because SPIN is not designed to deal with real-time systems. It is feasible to include some sort of constraints and counters but it comes to a high price in terms of modelling and efficiency. For the explicit consideration of time constraints within the specification of a Smart Home the reader may be interested in consulting [3] where the authors use UPPAAL [21], a model checker focused on explicit time-constraints.

4. Our case study: The Nocturnal Model

An example of a task oriented MAS model is the one used in the Nocturnal project whose functioning scheme is shown in Figure 3. The multi-agent system is conceived around the meaningful steps on the decision making of the system. Each important step in the decision-making chain is taken by a specialized agent which is sensitive to specific information collected through the sensors or generated by other agents. The modelling strategy we follow here is inspired in [22].

Activities of the client trigger sensors which are recorded as events in a database. These events feed to a group of monitoring agents specialized on night related situations (e.g. restlessness, bed occupancy and wandering). When the number of episodes of interest detected by any single agent is above an acceptable threshold, which is dynamically adapted to the client and the context, the agent involved contacts a coordinating agent (CA), which has a holistic view of the context, since it receives all the single agent’s reports. If appropriate, CA can trigger a therapeutic intervention with the aim of helping the client. If subsequent reports from the monitoring agents show there is still reasons for concern the coordinating agent can issue a new intervention or eventually, if the situation requires it, the call centre (in this project the call centre is located at Fold Hous-

[^4]: http://www.fastuk.org/research/projview.php?id=1447
ing Association) can be contacted so that a human deals directly with the situation. The system then is used as a way to increase independence with safety and to focus human interventions on those cases where they are really needed.

4.1. Nocturnal PROMELA Model

Figure 3 depicts the main actors and interactions within the Nocturnal system architecture. The model was conceived to explicitly represent those elements and relationships to test the idea and use it as a framework to experiment with and discover non-trivial features which escaped the initial analysis of the team. One of the versions of the model of the overall architecture is provided in Appendix A (see comments inserted providing explanation of the model). Each main element of technology and human actors depicted in Figure 3 is represented in the PROMELA model by a process type. Processes are autonomous entities running concurrently. Interaction amongst these elements is represented by message passing through synchronous channels. Naturally there are many features of the model that can be changed to experiment with.

The model we provide in this chapter uses a big number of channels as a way to achieve an output through the Message Sequence Chart (produced during simulation by SPIN) which explicitly highlights the agents being activated and their interaction with other agents or parts of the system.

Boolean variables have also been introduced to facilitate the verification of behavioural properties. As SPIN does not verify assertions involving local variables or channel operations directly we need to associate the changing values of global variables at strategic places of the model to capture the conditions we are monitoring.

The use of channels is expensive for verification but not for simulation. So this particular model based in channels will allow long simulations delivering a substantial amount of information about the possible combinations the system can go through. This benefit is counterbalanced at a later stage of the experiment with a high cost when we try to run different verification checks. This model should be taken only as a snapshot in
the lifetime of the system and as serving a specific purpose. There is no claim over this model being the best model or a final model for the system we are discussing. Different models can be built to obtain different benefits. This one serves the purpose to explain an idea and to illustrate the different typical pros and cons the developing team is exposed to in the typical decision-making process to be followed when applying verification methodologies and tools to a software system. Another approach would have been to seek a more economical representation of the concepts involved in the MAS system, but that would have obscured the model making it more difficult to understand, reason why we decided to give priority to pedagogical elements over efficiency.

4.2. Simulation of Nocturnal system

This model can be used for simulation in SPIN and several different types of views can be extracted as the simulation unfolds. Figure 4 shows the Message Sequence Chart produced by the messages interchanged by the different processes, each thread corresponds to a process. The threads as they appear in the figure from left to right corresponds to: client, environment, database, restlessness agent, bed occupancy agent (not appearing in this fragment of the MSC), wandering agent, coordinating agent, the therapeutic agent and Fold (where the call centre which deals with emergencies resides). The simulation shows a point where restlessness was recognized and the Therapeutic Intervention agent failed to calm down the person, forcing the system to contact the company (Fold), which in turn decides to contact the user directly.

Notice this model does not focus on frequencies but rather on possibilities, i.e. whether something can be achieved or not within that architecture. We have built separate models which specifically tackled lower level functionality, for example on how the monitoring agents can effectively keep track of the frequency of restlessness episodes, detect absence from bed or wandering, during a period of time.

4.3. Verification of Nocturnal model

Formally specified behavioural properties (e.g. \( []<>\text{activeClient} \), which means "infinitely often the client is active") can be explored using SPIN. These properties are usually related to the requirements of the system being examined. Examples of such properties for the Nocturnal case are: "Can the system monitor the client continuously?", "Are all sensor activations stored in the DB?", and "Is each emergency followed by a therapeutic intervention?".

As we explained before, readers who are keen on exploring the use of the model given in the appendix with SPIN will find that its use to run simulations of the system being modeled does not pose computational complexity problems, but verification can quickly push the resources of their machines to the limit. Those experimenting with the model will most likely have to modify the settings through the “advanced verification options” tab of the verification panel to increase the use of ‘physical memory’ and the ‘maximum search depth’ to as much as their machines have left after SPIN is loaded. Figure 5 shows that running a basic verification on the model of Appendix A is successful. It also shows the statistics of running the verification, amongst these notice the number of levels in the search tree is almost 70,000. This is due to all the potential combinations created by the boolean variables, channels, and PROMELA sentences in each of the processes, which generates a combinatorial explosion in the search space tree.
Figure 4. Simulation run of Nocturnal model
Figure 5. Statistics obtained at the end of a basic verification run.
To illustrate that not always properties have to be true to be informative and useful, we include a mix of properties and results, each one bringing us one step closer to the understanding of the system modelled. Let us consider the property: \( [] \text{sensorActive} \).

To check this property we need to add a variable that we can use to check whether we reached a specific position in the model, by adding a declaration of a variable (bool idle = true;) and then checking the property by including:

\[
\text{ltl p1} \quad []!\text{idle} \quad /\ast \text{i.e.} \quad []\text{sensorActive} */
\]

taking advantage of the definition of the process. Figure 6 shows the result. Of course, the property cannot be proved as the sensing equipment has been modelled in such a way they reflect the fact that real equipment is not permanently sensing (this could be because of absence of triggering events or because the equipment itself can malfunction, be occluded or run out of battery).

The counterexample shown in Figure 7 indicates it is possible for sensors to be in state idle. To make it more interesting, we set the variable idle initially to false so that it is not trivially satisfied.

A small modification to the previous property can be:

\[
\text{ltl p2} \quad <>!\text{idle} \quad /\ast \text{i.e.} \quad <>\text{sensorActive} */
\]

which yields a different result. This property can be verified to be true for this model as the process representing the behaviour of the sensing system (all sensors together) is such that it cannot be permanently idle if it is working properly, so eventually it will communicate that one sensor has been activated (see Figure 8).

Another property we can prove correct is that the restlessness agent detecting that a situation is interesting leads to the coordinating agent being informed:

\[
\text{ltl p3} \quad [](\text{RAInterested} \rightarrow <>\text{CAContacted})
\]

Before attempting to prove this property we need to slightly specialize the model to allow the check to take place. We can achieve this by adding the corresponding declaration for variables

\[
(\text{bool RAInterested, CAContacted};)
\]

and adding two assignments

\[
(\text{RAInterested}=\text{true}; \ \text{RAInterested}=\text{false};)
\]

inside the atomic declaration within the RestlessnessAgent proctype, as well as these two assignments CAContacted=\text{true}; \ CAContacted=\text{false}; inside the first atomic declaration within the CoordinatingAgent proctype (the one with the filter ra2CA?contact). The role of these assignments is to capture the states where the Restlessness Agent notices there is something of interest to communicate to the Coordinating Agent and also to capture the state where the Coordinating Agent takes notice of the Restlessness Agent message. With these additions now we are ready to ask SPIN to check whether the property formally stated above is consistent with the possible situations implicitly captured in the model. To check a property in SPIN we need to use
the verification panel where we can specify the property we want to check, this gives the result shown in Figure 9.

Just for the sake to illustrate another situation which is of interest, we can try to verify whether the following property is true (here TI is an acronym for "Therapeutic Intervention agent"):

\[
\text{ltl p4 } [\neg (RAInterested \rightarrow <>TIContacted)]
\]

Adding the declaration for variables

\[
\text{(bool RAInterested, TIContacted;)}
\]

and the assignments

\[
\text{(RAInterested=true; RAInterested=false;)}
\]

inside the atomic declaration within the RestlessnessAgent proctype, as well as these two assignments (TIContacted=true; TIContacted=false) inside the atomic declaration within the TherapeuticInterventionAgent proctype, then we are ready to check the property, which gives the result shown in Figure 10. But this is where we need to be careful and understand that the results when using a model checker should be always considered relative to the system modelled and to the language used to describe the situation.

A rushed interpretation of the formula above will be that every time the Restlessness agent detects a situation of restlessness, it will end up being dealt with by the Therapeutic Intervention agent. Here we need to consider this result with caution. The property is true with regards to the model we built but the result is not necessarily as strong as we will wish for the real system we want to build. What we really want to achieve is that the Coordinating Agent analyzes each situation and sometimes will decide to do nothing (which is indicated in the model through the \text{skip} option) whilst sometimes it will contact the TI Agent as a result. What we proved does not necessarily reassure us that the CA will contact the TI immediately as a result of an incident. If we look at the formula we verified, it only says that if the Restlessness agent is interested, then in the future the TI agent will be contacted, but the \text{<>} operator does not represent temporal immediacy. That is, \text{<>p} promises that a given proposition \text{p} will be true in the future at least once, but it does not tell us when exactly nor how often. So if we want to reflect in the strategy represented in this model that requests from the CA are taken with certain urgency or even that requests are considered in the order they arrive to the TI’s attention, then the model has to be specialized further.

This is the fundamental process guiding the use of models, they can be modified in different directions to explore the consequences of different strategies. This process is iterated as many times as required or feasible to inform the developing team before or during implementation.
Property 1 (\texttt{sensorActive}) is false
Figure 7. Counterexample for Property 1 (\texttt{sensorActive})
Figure 8. Property 2: `sensorActive` is valid.
Property 3 (\[\neg RAInterested \rightarrow <> CAContacted\]) is valid.
Property 4 \((\text{RAInterested} \rightarrow <>\text{TIContacted})\) is valid.
5. Lessons Learnt

The previous section has introduced a specific example of a system which has been designed using a multi-agent system and focused on tasks (agents are entrusted to detect and reason about specific tasks and problems which are at the core of the implemented system). More importantly, from this case study we can extrapolate a methodology or strategy to model similar task-focused multi-agent systems. The sketchy model shown below abstracts from the example itself and focuses on a generic task-focused MAS model, it is a sketch of a model in the sense it only suggests a strategy to follow.

/* Data Structures Declarations Section */
...
/* All synchronous channels, i.e. handshake guaranteed */
...

/* Process Declaration Section */

active proctype Human ()
{ end: do
  :: human triggering (or not) sensors
  od}

active proctype Environment ()
{ end: do
  :: read event inputs and pass them to DB
  od}

active proctype DB ()
{ end: do
  :: sensor reading event received -->
    decides which is the relevant agent to echo it to
  od}

active proctype AssistingAgent1 ()
/* detects one type of problem */
{ end: do
  :: reads relevant sensor -->
    if
    :: interesting --> contact Coordinating Agent
    :: ignore
    fi
  od}

...

active proctype AssistingAgentN ()
/* detects another type of problem */
{ end: do
:: reads relevant sensor -->
    if
    :: interesting --> contact Coordinating Agent
    :: ignore
    fi
od}

active proctype CoordinatingAgent ()
/* gathers advice from assisting agents and, when necessary, intervenes in the environment */
{ end: do
:: assisting agent 1 contacts coordinating agent -->
    if
    :: CA tries to deal locally with emergency
    :: CA tries to deal externally with emergency
    :: decides to ignore
    fi

... 

:: assisting agent N contacts coordinating agent -->
    if
    :: CA tries to deal locally with emergency
    :: CA tries to deal externally with emergency
    :: decides to ignore
    fi
od}

active proctype TherapeuticInterventionAgent ()
/* actuation in the environment to remedy a problem */
{ end: do
:: receives order to act -->
    intervention
od}

active proctype Emergency ()
/* acts when the system unsuccessful locally */
{ end: do
:: when requested -->
    delivers contact to caring circle
od}

This sketch of a model can be refined and adapted with the details of the specific problem in hand, the model we already analyzed and which is included in the Appendix provides inspiration on how to accomplish that final step.
6. Conclusions

We have explained the advantages of considering well established Software Engineering methods and tools to design more reliable AmI systems to be developed with multi-agent systems.

The current state of the art in verification is more oriented towards classical distributed systems, though it can also be useful for the analysis of MAS-based AmI systems, as we have demonstrated. However, the existing verification technology is not specifically intended for this type of systems. Hence, we propose using the current methodologies and tools until other approaches and means expressly designed for these systems arise.

There are some systems which can partially achieve the process we have illustrated with SPIN [23], and others which can cover the full process but (as with SPIN or UP-PAAL) not necessarily AmI oriented [24], still no system produces the whole process specifically tailored to AmI systems.

A slightly different approach for the specification and verification of agent models is presented in [25], which consist on checking properties against a limited set of simulation traces, instead of performing an exhaustive verification considering all the possible executions of the system modelled, as we propose in this chapter. However, it allows checking more complex properties, since the specification formalism used in the former to express the properties to be verified is more expressive than the one used in the latter.

As applications provided by AmI systems, such as automation, health monitoring and context-aware assistance, rely on a software architecture generally driven by a middleware, some requirements for designing and evaluating a middleware for these systems are proposed in [26]. While this work concentrates on this concrete aspect of an AmI system, our proposal is focused on verifying the main functionalities and features of its applications.

There has been attempts to obtain automatic translations from AgentSpeak (a specific well-known MAS notation) to SPIN, the experience showed this is feasible for a subset of AgentSpeak [27]. This of course does not mean that it is not possible to provide a complete automatic translation when we consider another MAS framework or another verification framework different from SPIN. Until such complete automatic translations are available, we can still rely on human experts to use verification frameworks to model and understand with the help of rigorous methodologies the AmI system under development.

Our chapter provides then a basic recipe, by way of illustration, on how to apply currently available verification technology to the development of more correct and safer MAS-based AmI systems.
References


[16] URL: www.acm.org/announcements/turing.html


A. Nocturnal Architecture PROMELA Model (extracted from [7])

/* Data Structures Declarations Section */

bool activeSensor= true;
bool contact= true;
bool takeAction= true;
bool intervention= true;
bool event = true;

/* All synchronous channels, i.e. handshake guaranteed */

chan client2sensors = [0] of {bool}; /* Client to Sensors */
chan env2DB = [0] of {bool}; /* environment to DB */
chan db2RA = [0] of {bool}; /* DB to Restlessness Agent */
chan db2BOA = [0] of {bool}; /* DB to Bed Occupancy Agent */
chan db2WA = [0] of {bool}; /* DB to Wandering Agent */
chan ra2CA = [0] of {bool}; /* Restlessness Agent to Coordinator Agent */
chan boa2CA = [0] of {bool}; /* Occupancy Agent to Coordinator Agent */
chan wa2CA = [0] of {bool}; /* Wandering Agent to Coordinator Agent */
chan ca2TIA = [0] of {bool}; /* CA to Therapeutic Intervention Agent */
chan tia2client = [0] of {bool}; /* Therapeutic Intervention Agent to Client */
chan ca2fold = [0] of {bool}; /* Emergency Notification */

/* ------ Process Declaration Section ------ */

active proctype Client ()
/* represents free will of human */
{ bool activeC;
end: do
:: tia2client?intervention -->
  if
  :: client2sensors!event
  :: skip
  fi
:: activeC -->
active proctype environment ()
/* generates sensor data and stores it in DB */
{ end: do
:: !idle -->
    atomic{client2sensors!event;
          env2DB!activeSensor;
          idle=true}
:: idle --> idle=false
od}

active proctype DB ()
/* stores sensor data and passes it to agents*/
{ end: do
:: env2DB?activeSensor -->
    atomic{if
    :: db2RA!activeSensor
        :: skip
    fi;
    if
    :: db2BOA!activeSensor
        :: skip
    fi;
    if
    :: db2WA!activeSensor
        :: skip
    fi}
    od}

active proctype RestlessnessAgent ()
/* detects restlessness episodes */
{ end: do
:: db2RA?activeSensor -->
    if
    :: atomic{printf("Restlessness Agent was interested
            on this information");
                ra2CA!contact}
        :: skip
    fi
od}
active proctype BedOccupancyAgent ()
/* detects out of bed episodes */
{ end: do
  :: db2BOA?activeSensor -->
    if
      :: atomic{printf("BedOcc. Agent was interested
          on this information");
        boa2CA!contact}
      :: skip
    fi
  od}

active proctype WanderingAgent ()
/* detects wandering episodes */
{ end: do
  :: db2WA?activeSensor -->
    if
      :: atomic{printf("Wandering Agent was interested
          on this information");
        wa2CA!contact}
      :: skip
    fi
  od}

active proctype CoordinatingAgent ()
/* gathers advice from agents and,
when necessary, intervenes in environment */
{ end: do
  :: ra2CA?contact -->
    if
      :: ca2TIA!takeAction
        :: ca2fold!takeAction
      :: skip
    fi
  :: boa2CA?contact -->
    if
      :: ca2TIA!takeAction
        :: ca2fold!takeAction
      :: skip
    fi
  :: wa2CA?contact -->
    if
      :: ca2TIA!takeAction
        :: ca2fold!takeAction
      :: skip
    fi
  od}
active proctype TherapeuticInterventionAgent ()
/* acts in env. to achieve goals set by coordinator agent */
{ end: do
   :: ca2TIA?takeAction -->
   atomic(tia2client!intervention;
             printf("Action Taken!"))
   od}

active proctype Fold ()
/* deals with emergencies */
{ end: do
   :: ca2fold?takeAction -->
           printf("Action Considered by Fold!"))
   od}