Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Research Report
RR-99-02

SIF - The Social Interaction Framework

System Description and User's Guide to a Multi-Agent System Testbed

Michael Schillo, Jürgen Lind, Petra Funk,

Christian Gerber and Christoph Jung

February 1999

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 2080
67608 Kaiserslautern, FRG
Tel: +49 (631) 205-3211
Fax: +49 (631) 205-3210
E-Mail: info@dfki.uni-kl.de

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel: +49 (631) 302-5252
Fax: +49 (631) 302-5341
E-Mail: info@dfki.de

WWW: http://www.dfki.de

Deutsches Forschungszentrum für Künstliche Intelligenz

DFKI GmbH

German Research Center for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest non-profit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focusing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialisation.

Based in Kaiserslautern and Saarbrücken, the German Research Center for Artificial Intelligence ranks among the important "Centers of Excellence" world-wide.

An important element of DFKI's mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

DFKI has about 115 full-time employees, including 95 research scientists with advanced degrees. There are also around 120 part-time research assistants.

Revenues for DFKI were about 24 million DM in 997, half from government contract work and half from commercial clients. The annual increase in contracts from commercial clients was greater than 37\% during the last three years.

At DFKI, all work is organised in the form of clearly focused research or development projects with planned deliverables, various milestones, and a duration from several months up to three years.

DFKI benefits from interaction with the faculty of the Universities of Saarbrücken and Kaiserslautern and in turn provides opportunities for research and Ph.D. thesis supervision to students from these universities, which have an outstanding reputation in Computer Science.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI's six research departments are directed by internationally recognised research scientists:

- ❏ Information Management and Document Analysis (Director: Prof. A. Dengel)
- ❏ Intelligent Visualisation and Simulation Systems (Director: Prof. H. Hagen)
- ❏ Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- ❏ Programming Systems (Director: Prof. G. Smolka)
- ❏ Language Technology (Director: Prof. H. Uszkoreit)
- ❏ Intelligent User Interfaces (Director: Prof. W. Wahlster)

In this series, DFKI publishes research reports, technical memos, documents (e.g. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster

Director

SIF - The Social Interaction Framework

System Description and User's Guide to a Multi-Agent System Testbed

Michael Schillo, Jürgen Lind, Petra Funk, Christian Gerber
and Christoph Jung

DFKI-RR-99-02

This work has been supported by Siemens AG, the Deutsche Forschungsgemeinschaft (DFG), and the Studienstiftung des deutschen Volkes.

© Deutsches Forschungszentrum für Künstliche Intelligenz 1999

This work may not be copied or reproduced in whole or part for any commercial purpose. Permission to copy in whole or part without payment of fee is granted for non-profit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Deutsche Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

ISSN 0946-008X

SIF - THE SOCIAL INTERACTION FRAMEWORK

SYSTEM DESCRIPTION AND USER'S GUIDE TO A MULTI-AGENT SYSTEM TESTBED

Michael Schillo, Jürgen Lind, Petra Funk, Christian Gerber
and Christoph Jung
MAS Group, DFKI, Im Stadtwald,
D-66123 Saarbrücken, Germany
{schillo, lind, funk, gerber, jung}@dfki.de
http://www.dfki.de/~sif
sif@dfki.de

Abstract

We present the Social Interaction Framework *SIF* and demonstrate how it can be used for social simulation. SIF is a simulation testbed for multi-agent systems. The key design aspects are the ability of rapid-prototyping, a broad implementation platform, the possibility of controlling agents by human users and easy access to the internal data of every agent in the simulation. SIF implements the EMS *(Effector-Medium-Sensor)* paradigm, which provides a generic agent-world interface.

In this document we describe the architecture, example applications that have been developed at DFKI and we give an easy to follow ten step guide for creating simulations with SIF.

Table of Contents

1 INTRODUCTION 3

2 THE ARCHITECTURE OF SIF 6

2.1 The Effector-Medium-Sensor Model 6

2.2 Information channels 7

3 THE KERNEL 8

3.1 The World Server 8

3.2 Agent 9

3.3 GUI 9

3.4 Control and Information Flow in SIF 10

3.5 Simulated Time 11

3.6 Blocking of Effectors 12

4 THE USER INTERFACE 12

4.1 Console GUI 12

4.2 Object Info Window 13

4.3 Control Pad 13

4.4 Scripts 14

4.5 Flags 14

5 TEN STEPS TO A SUCCESSFUL SIMULATION WITH SIF 15

5.1 Download 15

5.2 Set-up SIF 15

5.3 Set-up your programming environment 16

5.4 Designing the entities in your simulation 16

5.5 Designing the rules that apply in your simulation 16

5.6 Designing the state of the simulated world 17

5.7 How can agents perform actions? 17

5.8 How do they perceive changes in the world? 17

5.9 Implementation 18

5.10 Run your simulation! 18

6 EXAMPLE SCENARIOS 18

6.1 The Automated Loading Dock 19

6.2 Simulation of an Automated Trading system 19

6.3 Resource Distribution Agents 20

6.4 XSif – Search Agents 20

6.5 The Pursuit Domain in SIF 21

7 DISCUSSION 21
7.1 Conclusion 21
7.2 Outlook 22

BIBLIOGRAPHY 24

ABBREVIATIONS USED 25

APPENDIX A THE PACKAGES 27

APPENDIX B SELECTED CLASSES 29

Table of Figures

Figure 1 A screenshot of SIF using an example scenario 4
Figure 2 Using media for interaction 6
Figure 3 The effector-medium-sensor model 7
Figure 4 The parts of SIF that make up the simulation engine 8
Figure 5 An example agent in SIF 9
Figure 6 Overview on the information flow during a simulation 11
Figure 7 Menus and buttons of the graphical user interface 13
Figure 8 Using a control pad to control an agent 14
Figure 9 A screenshot of the loading dock simulation 19
Figure 10 The map of the agent when started (left) and after some exploration (right). 21
Figure 11 A screenshot of the upcoming SIF version 22
Figure 12 The directory structure of the SIF library 27
Figure 13 All classes of SIF and their inheritance and usage structure 31

1 Introduction

A recent development in Artificial Intelligence is *Agent-Oriented Programming* (AOP) (Shoham 1991), derived from Object-Oriented Programming (OOP): the entity in focus is a so-called *agent*, a unit able to autonomously plan and perform actions. Some approaches of AOP see communication between agents as a special case of acting, whereas others regard acting and communication as different forms of interaction. Tasks are performed in a distributed fashion through cooperation between agents

The *Social Interaction Framework* (SIF) provides a virtual testbed for evaluation and an environment for development of Multi-Agent systems. SIF's strength lies in the support of the design interaction intensive agent societies and its library of ready-to-use components for rapid prototyping. The framework can be adapted to a wide range of scenarios and types of agents, as it is divided into a kernel that implements the model of SIF and a toolbox containing ready-to-use objects.

In order to build reasonable simulations, the three strands of technology associated with Virtual reality (VR), Distributed Interactive Simulation (DIS), and Distributed Artificial Intelligence (DAI) have to contribute. DAI provides for the computational means to build (semi-) autonomous, interacting agent programs which are situated in an environment via well-defined perception and action channels. DIS researches the modelling of the (dynamic, kinematic) evolution of such an environment, including *virtual bodies* for agents. Finally, VR techniques deliver an integrative frame for visualising the simulated world and interfacing human users.

DAI is not only providing methods to build simulations, but is also, as a science, a potential user since it requires an intuitive framework for complex, distributed simulations. For example, issues of adaptivity and scalability in multi-agent systems (MAS) cannot be investigated without the use of large-scale simulations. Using a human-oriented, realistic model of the environment allows a suitable embodiment for evaluating intelligent agents. Indeed, other sciences, such as the social sciences, are as well becoming aware of the usefulness of co-habited computer simulations. A common architectural framework for simulation is thus reasonable and must satisfy the following requirements:

Human Users Interaction Simulations should allow users to survey and influence the state of the simulated world. An intuitive visualisation as well as a corresponding user interface is required. From an agent's standpoint, user-controlled avatars and agents should not be distinguishable. From a user's perspective, an avatar should be semi-autonomous, i.e., its lifelike low-level behaviour should be changeable by frequent user adjustments.

Broad Platform In general, simulations become the more attractive the more people they can reach, the more existing systems they can interface. Common and open standards, such as those based on JAVA™ , RMI, VRML'97 and CORBA are the key to connect users working on heterogeneous platforms and to connect agents built with differing tools and languages. Although the mentioned standards are not immediately designed with respect to the demands of simulation, the widespread interest in them by software companies and researchers will result in a similar effectiveness as home-grown, special-purpose solutions.

Autonomy and Transparent Distribution Agents (and avatars) are autonomous (semi-autonomous), i.e., they are able to freely compute their activity (in given bounds). In AI, this is usually described by the mutual interplay of two well-defined kinds of interfaces: by sensors and effectors. Here, the virtual world is responsible for "outputting" on the sensor interface as being fed by effector data. In turn, the effector data is independently derived by

the agent, i.e., in a different computational background, from "inputting" the sensor data (we elaborate more on these concepts in Section 2.1). When applied to a networked setting, this notion allows agents and avatars to safely reside and decide on local computers while perception and action are transparent mediators to the central, global world.

Fine-grained Action and Perception Very often, pragmatic and visualisation-oriented simulations provide every agent with the same abstract (qualitative) sensor and effector interfaces. This means that perception already incorporates high-level information of which a commonly agreed semantics is seldom to be found in an open context. Similarly, actions are coarse patterns which denote complex, but uncontrollable behaviour in the virtual world. Especially for evaluating intelligent agents in a scientific context, but also for creating more lifelike scenarios, different quantitative modes of perception and action, such as touching, smelling, looking, communicating, etc., should allow the agent to classify objects and outcomes of actions from its personal, characteristic perspective in order to choose from a richer space of behaviour. Realistic embodiment requires various of those *non-labelled* methods of sensing and acting at the same time.

Rapid Prototyping What makes virtual worlds so attractive at all stages of scientific research and development, is the possibility for rapid-prototyping. This is due to the fact that the granularity of the world is not determined in advance. Thus the designer can incrementally refine the level of abstraction and therefore gradually approximate real-world characteristics, such as continuity, dynamics, non-determinism, and inaccessibility, etc. as it may be required by the actual experiments. A simulation architecture should support this process.

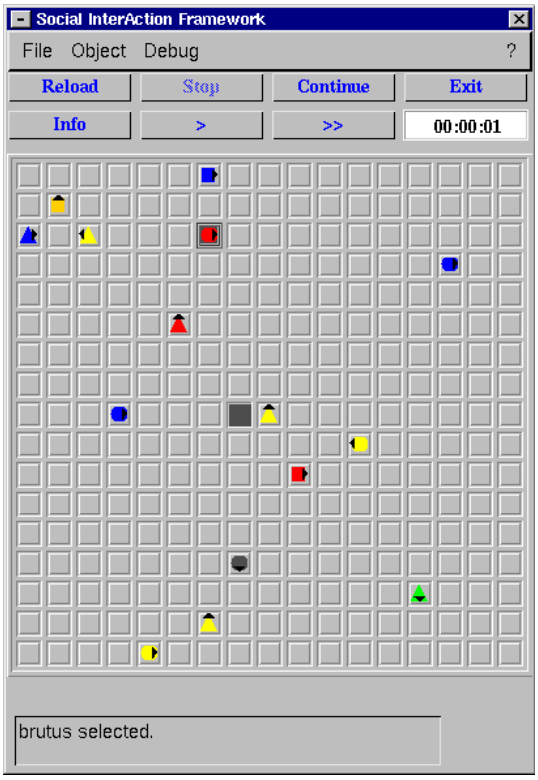

Figure 1 A screenshot of SIF using an example scenario

Our solution to this list of desiderata is SIF. SIF is implemented in JAVA™ JDK 1.1. and has interfaces for agents and visualisations using other languages (e.g. VRML™). All agents and the medium run in separate JAVA™ threads to provide realistic assumptions on parallelism in the scenario. Figure 1 shows an example of a SIF scenario. The scenario consists of a simple grid representation and a group of agents represented by geometric

shapes. The objects that were used for this example are part of the toolbox of SIF and can be adapted to new scenarios.

The SIF design and functionality has been in parts inspired by the ongoing debate about agent architectures and the interaction mechanism between them. SIF is yet another agent testbed, but has many different aspects, compared to more traditional systems. Testbeds for agent architectures provide simple and effective tools to investigate and explore several aspects of agent architectures with controlled experimentation (Hanks et al. 1993). Experiments with pre-defined settings help to compare performance of different agent architectures in various rich environments.

Of the broad variety of testbeds, that have been designed and implemented, we focus here on a small selection of related or orthogonal systems. One of the most well known testbeds within the Artificial Intelligence community is the TILEWORLD system by Pollack and colleagues (Pollack and Ringuette 1990). The TILEWORLD system provides a grid-like environment. The task of the agent is to collect tiles and deposit them in randomly occurring holes in the grid. Agents have limited resources in terms of carrying capacity and lifetime. They also have to consider changes in the environment, due to random appearance of holes, which have only a certain lifetime as well as random appearance of tiles. Sensors are not explicitly represented, the agents can always access the environment and its changes. Some of these ideas can also be modelled in the SIF system: in the displayed grid world scenario, agents are situated in a dynamically changing boxed environment. While TILEWORLD agents have a fixed set of tasks, agents in the SIF world can be provided with much richer goals as well as confronted with different dynamics in the environment. Depending on the user's goals and skills, she can design the environment to her experiment needs. Also, SIF provides the user with a smooth application programming interface to the EMS paradigm, which comprises a unified model for sensing and acting in an environment. There are several other grid world agent testbeds, e.g. NASA Tileworld (Reference to Phillips and Bresnia 1991), which was developed independently from TILEWORLD or the MICE simulator (Montgomery et al. 1992).

Other agent testbeds were developed for different environments. The fire fighting domain inspired the *Phoenix framework* for implementing and testing multiple autonomous agents ((Hart and Cohen 1990), (Greenberg and Westbrook 1990)). The system consists of three components: the simulator, the environment and the agents. The *simulator's* tasks are to maintain and update the map, to synchronise the activities of the environment and the agents, and to gather data. The Phoenix *agents* sense and change the Phoenix environment by sending messages to the *simulator*. Exogenous events like e.g. wind are also implemented and can cause fires to grow. Phoenix agent have limited sensors and perceive their environments through messages. Rather than implementing an abstract task environment, Phoenix aims at providing a realistic simulation of a single domain. The fire fighting domain is a domain still favoured by multi-agent researchers (Wiering and Dorigo 1998).

Another effort to model a domain and use the system as multi-agent testbed, is the *Truckworld* (Hanks et al. 1993). The system has been designed to test theories of reactive execution and to provide motivating examples for theory of reasoning about dynamic and uncertain worlds. The major goal was to provide a realistic model of the domain accessible for the agents, but without physical sensors. Nevertheless, Truckworld simulates sensors, such as camera reports for object features, sonar reports for obstacle detection or scales reports for weight detection. The Truckworld domain is a road network and the trucks travelling along the roads are agents. Weather influences the road surface as well as truck behaviour. Agents can communicate via radio or loudspeakers.

The MYWORLD system by Michael Wooldridge (Wooldridge 1995) has been especially designed for BDI agent architectures. The testbed focuses particularly on BDI agents and the

core of the system is the underlying BDI logic and how the interaction of the agent with its environment can be described with such a logic.

When designing the SIF system we did not have in mind to model a complex domain in great detail, such as the Truckworld or Phoenix do. Rather, we wanted to provide an abstract model of interconnecting agents through strongly restricted channels to the environment in which they are situated. Since we introduced the notion of medium as the description of the world in terms of physical laws, it is possible to design experimentation environments in great detail. Depending on the users skills and needs, she hooks her agents provided with her favourite agent architectures into a complex environment defined through the medium.

This document is structured as follows: In the next section we describe the model that underlies SIF. For a better understanding of how SIF works, Section 3 will give an introduction to its kernel. To pave the way to using the system, Section 4 describes the complete user interface. After this overview on the possibilities with SIF, Section 5 gives a step by step guidelines on how to implement a new simulation with SIF. In the appendices we include more information on the source code.

More information on SIF including a full description of the source and all interfaces can be found at www.dfki.de/~sif.

2 The Architecture of SIF

2.1 The Effector-Medium-Sensor Model

A very general and broadly accepted definition of an agent is provided by (Russell and Norvig 1996):

> *An* agent *is an entity that can be viewed as* perceiving *its environment through* sensors *and* acting *upon that environment through* effectors.

The so called *Effector-Medium-Sensor paradigm* (for short EMS) reflects this definition (Lind 1998). This paradigm provides an appropriate abstraction of a (human) agent acting and interacting with its environment and other (human) agents. For example, when humans talk to each other, a speaker will use the speech apparatus (effector) to set the surrounding air (medium) in vibration. In a more detailed picture, the effector will influence the state of specified features of the medium, which depends in our example on the content of the speaker's message.

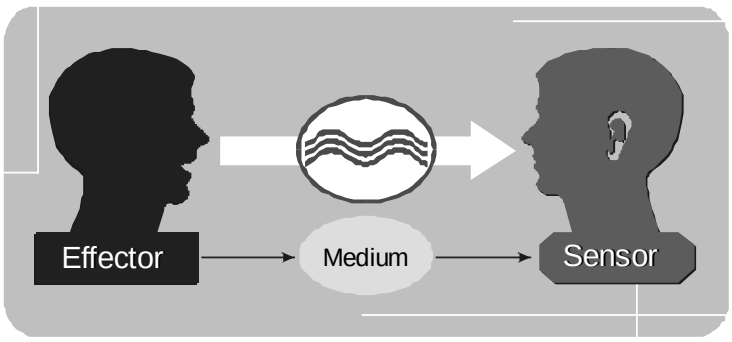

Figure 2 Using media for interaction

The medium itself connects features to a continuously updated structure, which could compute, in our case the propagation of the vibrations caused by the act of speaking. Finally these vibrations will then reach the receiver's ear, a sensor, which is connected to particular features of the medium and is able to transduce their state into an internal sensor reading (Figure 2). In the SIF architecture the medium is in contact with a range of different sensors and effectors used by the agents.

The *effector-medium-sensor model* is the core concept of SIF. It pinpoints the major features as there are: multi-threaded implementation, asynchronous data flow and subject-oriented point-of-view. In the centre of the model is the *medium*, which controls the simulation. It transfers information from and to the agents via *sensors* and *effectors* (see Figure 3). Sensors represent the input devices of an agent, just as the effectors represent its output devices. The data transferred is called *percept* and *action* respectively. Note that sensors do not observe necessarily everything that happens in the world, but can be limited to certain aspects or a specified range.

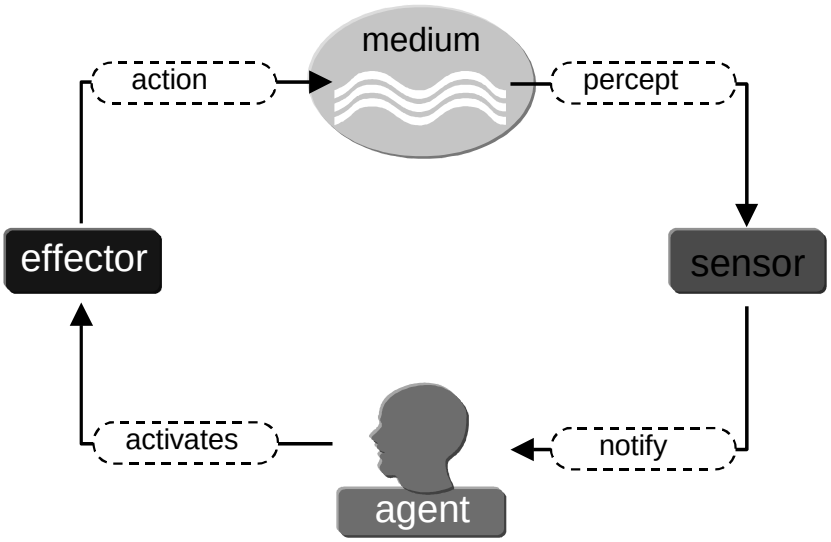

Figure 3 The effector-medium-sensor model

The agents in the model act independently from each other and choose the effectors that will produce the appropriate action. An effector is aimed at changing the world. When triggered, this effector sends an *action* to the medium, which modifies the representation of the world according to the type and the parameters of the action. Now it computes sensory input and sends it as *percepts* to all sensors affected. This flow of information is shown in Figure 3. The reason for this apparently complicated way of computing new states of the world is that the medium is supposed to be the only carrier of information and consequently, the subjects in the world can access information only via the medium.

2.2 Information channels

The sensor-medium-effector model is used consistently in SIF, i.e. the graphical user interface (GUI) is linked to the other parts like an agent. Thus all agents communicate only via the media, not directly with each other. This holds for all actions including communication, since we view communication as a special kind of action. To illustrate the communication flow through the media, assume that we have two Agents *A* and *B*. *A* wants to send a message (a communication action) to *B*. To do this, *A* activates its communication effector, which sends this action to the medium. The medium evaluates this action, identifies

the receiver and sends the message as a communication percept to the communication sensor of *B*. Note that the agent must explicitly check the success of an effector activation. The way to do this is to implement a sensor that tracks the agent's movement (similar to a global positioning system) or to implement a sensor that observes the change of the visual data and then to interpret the data of these sensors. For example, the activation of a *move effector* in front of a wall will not cause an error message. Instead, it is the task of the agent to find out that its current position has not changed.

The flow of information through the media is also characterised by the application of time (see also Section 3.5). Every action is assigned a certain amount of simulated time that it will take to complete this action. For the completion of an action this means that the environment will be checked whether the preconditions of the action hold, while the action is waiting to be processed. During this time the effector that generated it, is blocked (or "busy") and cannot generate other actions. This does not stop other agents from generating actions in the mean time.

3 The Kernel

In this section we give a more detailed description of the main parts of SIF, how they work and interact. Additional information can be found in the Application Programming Interface (API) which is available at the SIF homepage.

3.1 The World Server

Files: `WorldServer, Medium, SIFWorldRepresentation, ObjectInformation, PositionContents`
In package: `src.kernel` and `src.AOE` for the Medium.

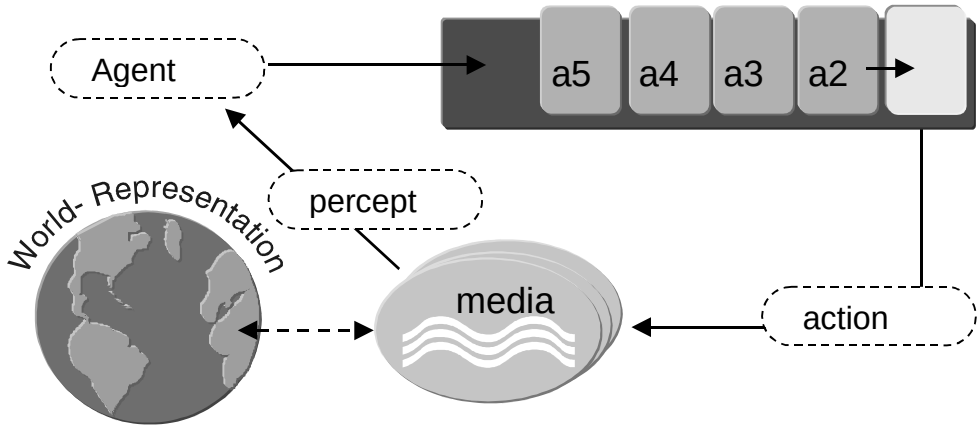

Figure 4 The parts of SIF that make up the simulation engine

The world server is the object that all agents, effectors and sensors connect to. The world server passes their reference on to the media that the effectors and sensors will deal with. The media update the world representation and organise the flow of information during the simulation. The world server stops and starts the simulation and provides debugging and data collection facilities. The media on the one hand define operationally the state transition of the world for every action. On the other hand they processes the data in the world representation to suit the specifications of the sensors according to the scenario. An implementation of the abstract class `SIFWorldRepresentation` defines a data structure, which carries information on position, appearance and capabilities of objects in the environment, the

specifications of the environment itself and the services needed or connected to these objects. The interaction between these parts is displayed in Figure 4.

For creating individual scenarios, it is necessary to override methods of the class `Medium`. The media are executed in the same thread as the world server when a simulation is running. The contents of each co-ordinate in the simulated world is represented by a data structure called `PositionContents`. Additionally, it stores a boolean value telling the GUI whether this position should be marked or not. This is used for visualising the selection of an object by the user. The information on the object itself is stored in an instance of the class `ObjectInformation` which contains data on how to access the object, its co-ordinates, etc.

3.2 Agent

Files: `Agent.java` and derived classes
In package: `src.GOL.agents`

Figure 5 shows an example agent with effectors and sensors. As demonstrated here, an agent can have different types of sensors and effectors in any number. The effectors could be for instance, effectors for moving, communicating, turning a robotic arm, etc. Sensors could be for vision and communication. In order to implement an agent, derive a class from the abstract class `Agent` and provide sensors and effectors.

Figure 5 An example agent in SIF

The agents, the GUI and the SIF kernel run independently from each other in a JAVA™ threads of their own. In the current implementation the agents must not commit more than one action per activation. This is necessary to by-pass the lack of pre-emptive scheduling for some platforms in JDK 1.1. As soon as a new JDK without this restriction gets published, this obligation is void.

3.3 GUI

Files: `SIFGUIAgent, SIFCanvas, SIFScriptEditGUI, SIFControlPadGUI, SIFObjectInfoGUI`
In package: `src.GUI`

As mentioned before, the GUI is linked to the core of SIF just like any other agent, i.e. it communicates via sensors and effectors. The percepts that it receives tell the GUI to update certain parts of the visualisation, etc. This is especially true for the `SIFGUIAgent` (which is an abstract class) or the `GUIAgent` class respectively. A special part of the GUI, the `SIFControlPadGUI` (as well as its implementation the `ControlPadGUI`) is capable of forcing agents to perform certain actions. This is done by sending an action to the world server that contains the information that the user wants the agent do something. This action is

then forwarded to the agent. All this is embedded in the EMS-model of SIF. For easier testing and varying of experiments in the simulated world, SIF contains a mechanism to specify simulations in a simple script language. These scripts can be loaded, modified and reloaded. The ScriptEditGUI enables the user to alter such scripts and restart them without need to shut down the SIF system. Any text editor can be used to modify the scripts.

The GUI runs in a JAVA™ thread of its own. So in total there are 2 + n threads running, n being the number of agents.

3.4 Control and Information Flow in SIF

Due to the EMS-paradigm the flow of control and information in SIF is rather simple. A detailed overview is given in Figure 6. To explain the information flow we assume that the simulation has been running for a while and a certain agent (upper left box in Figure 6) is about to choose its next move to make.

The decision making process takes place in the `act()` method of the agent (this will be changed with the next new release of JAVA™ JDK, see Section 3.2). As soon as this method has made its choice, it activates the corresponding effector. If an agent has decided to move forward, this could be an effector that controls the motor of a real world robot.

In the simulation, the effector wraps the information of what is intended to happen into an object called an *action*. The world server stores all incoming actions in an event queue. The actions are processed first-in-first-out. The world server passes the first action in the queue to the medium that will deal with this action (the world server finds the correct medium by calling the `getMyMedium()` method). The medium then accesses the world representation to find out whether the action is acceptable. If this is the case it will adjust the state of the world in the world representation and send a percept to the GUI, telling that a part of the world has to be redrawn and commit the agents move. The latter is pictured by the arrow pointing from the media to the GUI. Otherwise the medium will continue by computing new input for all sensors that are affected by this change of the world. The input is sent in form of percepts (with a subjective viewpoint) to the agents.

Now one cycle for the agent is completed and it can perform the next action. Please note that a side effect of the asynchronous EMS-model of SIF is that the agent does not have to wait for this feedback from the world. It can also start all sorts of actions, hoping that they will be suitable by the time they are being executed (see the Section 3.6 on this).

The last part of the figure to be explained is the arrow pointing from the GUI to the media. On this path information flows in two cases: when the experimenter uses a control pad (which is part of the GUI) to force a chosen action upon an agent or when the experimenter wants to start or stop the experiment. In the case of using a control pad, a command is sent to the queue in form of an action. Such commands get passed on to the agent and force it to an effector activation according to the command. The other case takes place if the user wants to state of the simulation itself, e.g. if a command is send to start or stop the simulation. Then the command will be redirected by the media to the world server‘s action management to start or stop the control flow. In the implementation we use a separate queue in the world server to store actions coming in from the GUI. This queue will be processed before the agent queue is being looked at.

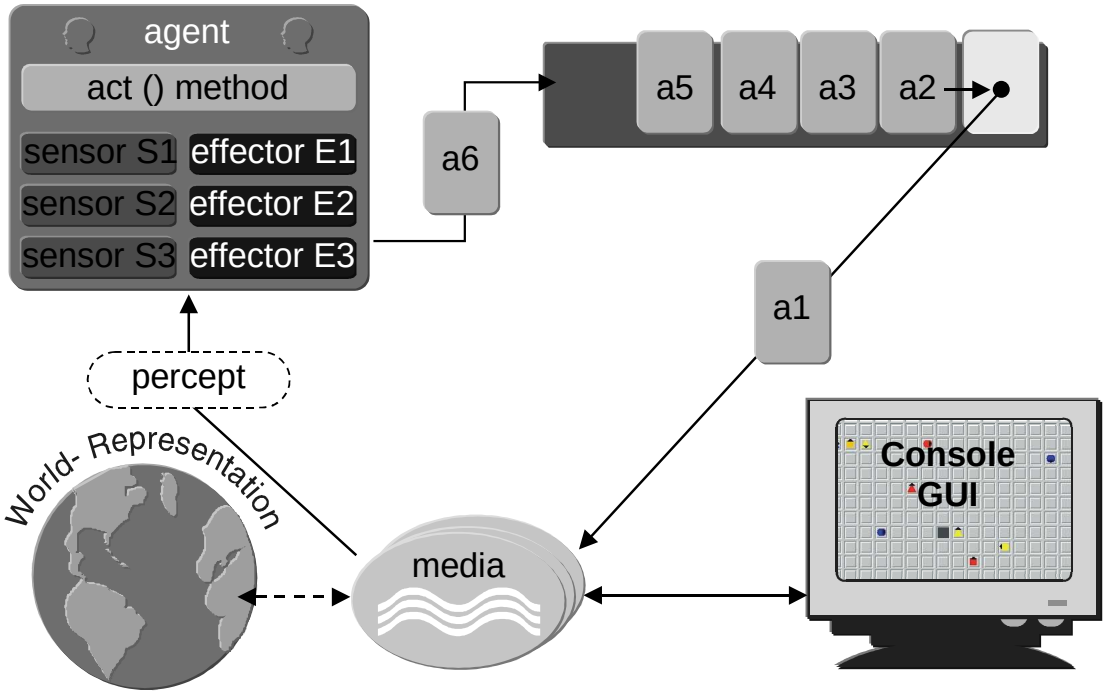

Figure 6 Overview on the information flow during a simulation

3.5 Simulated Time

SIF is equipped with a mechanism that simulates time. This is implemented by using a separate agent that runs in the background. Whenever this agent gets processing time, it will send an action to all media like a clock[1]. For observing the simulation, the current time is displayed by the GUI. The media can be intercepted this action in their `commitAction()` method to model time-dependent changes in the world representation. For example, the increase of some resource can be modelled by incrementing a variable in the world representation whenever the medium receives a new message from the timer agent.

The fact that the timer agent (or clock) is part of the simulation guarantees that the processing speed of a sequence of actions will always be the same in terms of simulated time. If the system runs faster because the virtual machine can allocate more resources to the simulation engine, the timer agent will also get more resource.

An example for the need of simulated time would be a number of actions that have different processing times. If an action gets sent by an effector, the world server will check how long this action would take to complete[2]. The world server will calculate the time when the action should be finished and uses this as the key to sort the action into the actions in the queue. As we have already mentioned, the world server will always process the first action in the queue. So it will process the action if all actions that have shorter durations or have been posted earlier are processed. This does not mean that an agent can block other agents that post long-term actions, because the timer agent also progresses so eventually the current time will be greater than the completion time of the long-term actions.

1 In fact, the action is called `ClockAction`.

2 It will check a field that the action inherits from the abstract class `src.AOE.Action`.

Whenever the world server has processed an action it checks if the remaining actions may stay in the queue. In other words, the world server checks if the preconditions of the still to process actions still hold[3]. Only if the preconditions for an action have always been holding during the action was in the queue, it will be committed to the state of the world. For an example of such an action try the script `time.script` when starting `OurWorld`. The yellow agent can only jump over the red one, if red one remains on its position. If it leaves this position, the action from the yellow agent gets removed from the queue.

3.6 Blocking of Effectors

By default, SIF is configured not to let an effector send an action as long as there is still an action of this effector in the queue. This will help to keep the number of actions that are waiting to be processed small. It will also make some of the behaviour of the agents more natural: An agent cannot pick up something while already picking up something else, it needs to wait until the first action is complete. Furthermore, it can be defined that agents are not supposed to perform two things at a time. For example, two effectors can be defined as mutually blocking to prevent an agent from turning while it is moving forward. Implementation details can be found in the class documentation[4].

During simulation the evaluation if an effector is allowed to post another action naturally takes time. This check can be disabled by starting SIF with –n (see Section 4.5 for more information).

4 The User Interface

After starting SIF a graphical user interface (GUI) appears on the screen; very similar to the one pictured in Figure 1. In this section we give a description of the interface and its functionality.

4.1 Console GUI

Buttons: The console for SIF as pictured in Figure 7 is equipped with seven buttons. All buttons will only be inactive if their functionality is not available in the current context. Two buttons let the simulation engine start and stop; the two buttons marked with arrows let the engine work in single step and fast single step mode. Fast single step means that five times as many actions are performed as in the single step mode. The *start button* will turn into a *continue button* as soon as the simulation has been started. The *reload button* will kill all objects and the world representation from the simulation engine and reload the previously selected script. The *information button* will start a window with information on a selected object (therefore an object or agent must be highlighted by clicking on it first). The same functionality is activated when *double-clicking on an object*. In the second row of buttons on the far right there is a display of the current simulation time.

Menus: The *file menu* offers general functions like loading a new script or saving the current state of the world as a script (see the Section 4.4on scripts). Scripts can also be edited in order to modify the simulation. The *object menu* allows to kill marked objects and display information on an object (which is the same function as the info button). The *debug menu* helps to check if the simulation is running correctly. The thread control flow can be checked (in order to see if the agents are being activated by the system) and display the control pad,

3 This is done by calling the methods of `src.AOE.Action` for precondition checks that should be overridden when using processing time other than 1 in a simulation. Check the html documentation for more details.

4 Look at the html documentation on the class `src.AOE.Effector`.

which allows to send messages to an agent during run-time[5]. Three further options let the user dump the contents of the agent table, the world representation and the action queue of the world server.

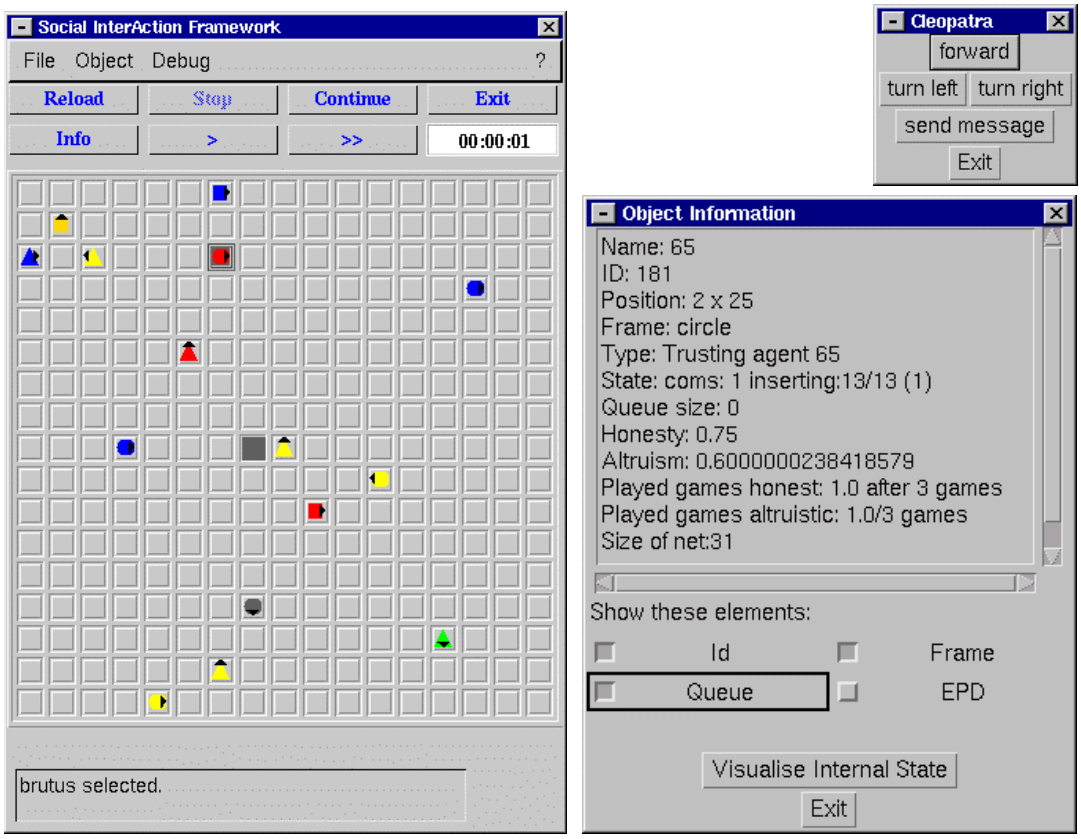

Figure 7 Menus and buttons of the graphical user interface

4.2 Object Info Window

For every agent and object, this window (see bottom of Figure 7) will display some general information like it's id, name, type and its position in the simulated world. Also it displays the last action of an agent, the current size of its percept queue and a string describing its current state[6]. For the agents that come with SIF we provide visualisers for further information e. g. on the sensors: which range they have (the corresponding area in the grid will be shaded) and what they detect at the moment. They are being activated by clicking the "show internal state" button. Any visualiser will automatically be linked into this window.

4.3 Control Pad

The control pad (at the top of Figure 7) lets the user send messages to agents. These agents must provide a sensor for such messages. Try the script `standard.script` when starting `OurWorld` and activate the control pad when agent "Brutus" at position (7,3) is marked. The user can make it move, turn or send messages.

The flow of such a control flow is depicted in Figure 8. The flow of control uses the EMS paradigm to route the information from the user to the agent to be controlled. To fit into the paradigm, the user must send his or her information by using an effector. This effector is

[5] For more information on the control pad see Section 4.3

[6] See the class documentation of the class `SIFObject` for more details.

activated by the control pad. The information gets send to the medium which can test if the user is allowed to send actions that influence an agent. It will infer which agent to send the message to. The agent to be controlled will receive a percept via a sensor about the command of the user. If it is programmed to react to such percepts it will activate its effectors accordingly. After the media have processed their actions, the user will recognise their effects by the visualisation of the data that is being received by the sensors of his or her GUI.

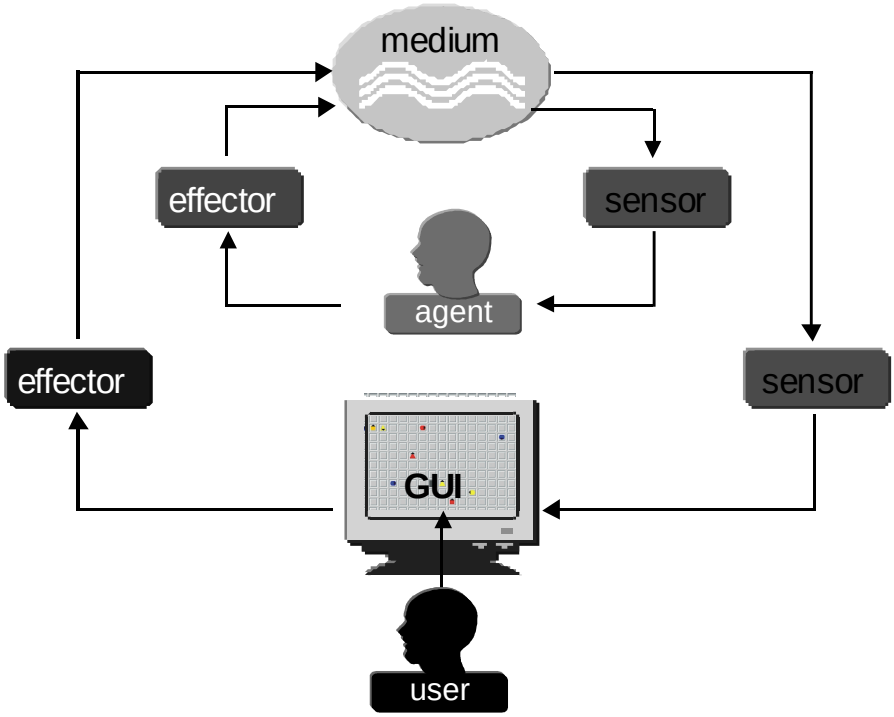

Figure 8 Using a control pad to control an agent

4.4 Scripts

SIF allows a simple script language to specify experiments. This language allows empty lines and comments (that is, lines that start with „#“ will be ignored during parsing). Any scenario specific definitions of agents, objects and the world representation should be defined in a class that inherits from `src.kernel.SIFIncorporateScript`. For a reference implementation, take a look at `example.src.IncorporateScript`.

4.5 Flags

SIF can be started with a number of flags:

```
java OurWorld [–a] [-b] [–n] [-o file][-s file][-v]
```

-a The simulation engine will start immediately without waiting for the user to press the start button. This is useful if using SIF in batch files in combination with the –s option.

–b Lets the user run a benchmark test. This option needs the file `benchmark.script` to be in the directory where the `OurWorld` class has been started. It will help the user to find out in which system configuration the simulation runs best.

–n Turns of the function that effectors can block each other. Using this option may speed up SIF, but it can also result in loss of performance if mischievous agents block the system

by exhaustive use of producing actions (see Section 3.6 for more information on the mutual blocking of effectors).

-o Lets the user specify a script where SIF should write its output to. This will only work in combination with the –v option and with the output that is generated using the methods mentioned there.

-s Specifies a script to be loaded on start-up. This means that the user does not need to choose a script from the list whenever SIF is started, which is especially useful during debugging.

-v Causes SIF to be verbose. We recommend not to use the `System.out.println()` method of Java™ but the `log()` method that every `SIFObject` inherits from the class `src.kernel.Basics`. This will help to find out which of the objects has produced the output and the user can also have a print out of the current stack of the virtual machine if `logError()` is used[7].

5 Ten Steps to a successful Simulation with SIF

This section guides for the experimenter through all the steps required for conducting simulations with the social interaction framework. For the sake of convenience we address in the following the you as the reader directly. We will start with the download and setting up of the files (steps 1 to 3). Then we will assist you in preparing the implementation of your simulation step by step (steps 4 to 8). This preparation will provide you with a set of concepts that will be easy to implement (step 9). Every section will conclude with a description of the result of each step that will help you to keep track of the progress of the development of your simulation. If you encounter any problems during download, set-up or implementation, if you have any bug reports or feature requests, please do not hesitate to contact the developers at `sif@dfki.de`.

5.1 Download

Create a directory named `sif` in your workspace. Start your web browser and go to the homepage of SIF which you can find at URL http://www.dfki.de/~sif/. Go to the download page and you will find a link *download now*. Click this link and save the file in your SIF directory. If you would like to have access to the source code, unpack the file and feel free to browse it.

→ **You should now have a directory `sif` that contains a copy of the SIF JAVA™ archive file `SIF.jar`.**

5.2 Set-up SIF

For setting up SIF you need an operating system that supports JAVA™. This could be Windows, MacOs, Linux or Unix. SIF has been designed for JDK 1.2, but it runs just as well with JDK 1.1. If you have downloaded SIF, set your paths according to the location of your SIF System. This means you need to set your classpath variable. For example, if you put SIF into a directory `/home/username/sif` or `c:/sif`, you set the classpath:

```
Setenv CLASSPATH /home/username/sif, or
Set CLASSPATH=c:\sif
```

[7] See the class documentation for `src.kernel.Basics`. Take also a look at the methods `toString(int)` and `printStack(String)`. These methods are useful in way that you will hardly need a debugger.

respectively for Linux/Unix or Windows

If you use an IDE such as Symantec Café ™, CodeWarrior ™, etc. then you can either include it as a new project or just set the path for system files to your SIF directory.

→ **You should now be able to run the following command and see an example implementation of SIF. In the directory example/scripts you will find a range of scripts, select for example `standard.script`. The simulation engine will then load this script and prepare the simulation so that you can run it[8].**

```
java example.src.OurWorld
```

5.3 Set-up your programming environment

If you already have a satisfactory programming environment, you can skip this section. However, if you need to find an environment for development with Unix, we recommend you use XEmacs. There are a few tools available on the download page that we consider very useful to you. As compiler we recommend Jikes ™ which is a freeware compiler produced by IBM. It is an incremental compiler and a lot more efficient than javac. However, you should try javac from time to time on your sources because under certain circumstances Jikes may have problems with linking classes if the inheritance structure is changed very often in one session (this does not happen very often). The question is that none of these two compilers is better in the sense of the word, but they have different strengths. Try both and you will easily see how you can use them together more efficient.

If you use XEmacs you can apply its java-mode and an application we provide: srcguiInfo. This application helps you to organise your project if you do not want to use a commercial IDE. Start srcguiInfo with the two parameters java and XEmacs (as described on the tips & tricks web page) and you will find navigation through your source files a lot easier. Together, XEmacs, Jikes and srcguiInfo make a fast and easy to use developing environment at no cost.

→ **You should now have an editor and a compiler ready for use.**

5.4 Designing the entities in your simulation

Now lets start to create the simulation! We assume you can describe the intended model in the terms of the concepts that appear in this and the following sections.

One part of the simulation will be actors i.e. units that will have a *behaviour* of some description. We will call them *agents*. Make up a list of the types of agents you identify in your setting and what *actions* each must be able to perform. Most likely, you will also have other units that will not be able to do something, but will be passive (walls, boxes, etc.). We call these *objects*. Later on, when we discuss the inheritance structure of the JAVA™ code you will find that agents are derived from objects. This means that agents are special objects in the sense that they are objects that can apart from just being in the world, affect the world.

→ **You should now have a list of types of agents and objects that populate your simulation. You should also have an exclusive list of actions that are possible in your simulation**

5.5 Designing the rules that apply in your simulation

Now you need to find out what the principle rules of your simulation are going to be. That is, describe exactly what should happen in the world if one of the actions is performed. You can think of these rules as the natural laws of the simulated world. Write down what the

[8] For an explanation of the buttons and menus of SIF see the Section 4.1

preconditions of these rules are. These define when the rules can be applied (and when this is not possible).

→ **You should now have a list of rules that define what happens if a certain action is performed. In your list of actions that are possible you should have a description of the preconditions of every action.**

5.6 Designing the state of the simulated world

When making up the previous lists, you may have encountered the problem that you wanted to refer to some variables that describe the state of the world. For example you would like to represent positions on a plane or in a room, so you would need some sort of an array data structure. Gather all these variables in a list for the *world representation*.

→ **You should now have a list of variables that you need to describe the state of your simulation**

5.7 How can agents perform actions?

In SIF we used the EMS-model which in short means that agents do not perform actions directly but they have *effectors*. Examples are effectors that make the agent move one step forward, turn it around, pick up an object, transfer money to another agent, etc. You need create a new effector for every action. Some actions are similar, in the sense that they affect similar variables of the world representation. Group these actions so that for each group there is one effector (this will make implementation easier).

To put more structure into the implementation, we would like you to think about a second part of the EMS-model, the *medium*. Media take actions, compute the change of the state of the world and transform the actions into something that can be perceived by other agents. Media tackle certain aspects of reality; e.g. air is responsible for the transport of sounds, light for seeing, etc. In other words, media implement the rules of what happens after an action has been performed. You can now easily see that they represent exactly the list you already made up in Section 5.5. Together with this list, we want you to group the effectors by the rules of the simulated reality they affect. A straightforward partition could be to have a medium that deals with communication and a medium that deals with moving objects in the world. Every effector must be assigned to one medium only (otherwise the partition will not gain any performance).

→ **You should now have a list of effectors, a description of what they can do and for each effector a list of actions that should be performed by this effector. You should also have a list of media and for each medium the set of rules from Section 5.5that it implements.**

5.8 How do they perceive changes in the world?

Just as in reality our agents do not perceive the world as it is. Humans do not really see light, but receive sensor data from their eyes and associate this data with stored experiences. To enable our agents to react on changes in their world we will something similar and design *sensors* for them. This is fairly simple, as most of the sensors will correspond in some way to the effectors. When you make up a list of all sensors needed by the agents, you may discover, however that there are sensors that observe things your agents cannot effect e. g. time. Another important fact in the context of the agent's actions is that an agent will never learn about the success of an action, unless it has a sensor that is designed to do so. For example, if an agent moves forward, it will not know whether it actually moved forward unless it has a sensor that tells it exactly this or it infers it from other sensors.

The data the agents will receive through sensors is called *percepts*. For an elegant design of your simulation be careful not to use any other information sources for agents than sensors. In particular, do not use direct references to other objects in the world or direct access to the world representation. Always filter such an information through media that clone the data from the world representation and wrap them into precepts.

→ **You should now have a list of sensors and what part of reality they watch. This implies that you know the media they will connect. For each agent you should know which sensors it will need.**

5.9 Implementation

Now you can start implementing your simulation! According to the data collected in the previous steps, you can now write JAVA™ program code. You might want to check first which of the objects that already come with SIF are useful for your simulation, so we recommend you explore the examples and take a look at Section 6.

Write one class file for each agent, sensor, effector, action, percept you need. Implement the media and the visualisation (if they differ from what is already in the object libraries of SIF). We recommend you arrange your files in a directory structure that reflects the structure of SIF (see Section Appendix A).

To get SIF running with your implementation you need to override the class `WorldServer` (`src.kernel`) and create objects for the world representation, graphical user interface, etc. This is rather easy, just cut and paste these lines of code from `example.src.OurWorld`. Name the file after your simulation, it will be the class that you will start to start your simulation. When started, SIF will always let you choose scripts from the directory where you started it.

If you need help on how to optimise development, feel free to check the tips & tricks web pages of SIF.

→ **You should now have a collection of classes that inherit from code out of `sif.*` packages. This code will be easy to move from one platform to another.**

5.10 Run your simulation!

Start your simulation with the class the overrides the `WorldServer`. It will then prompt you for a script to run. Choosing one and pressing start on the GUI will start your simulation. If you feel that the system is too slow on your machine check our tips & tricks web page maybe there is an easy way to speed things up.

→ **You should now have a simulation that can be easily transferred to other platforms and be used by a great number of users.**

If you encounter any problems during this process or if you have any feature requests for SIF, please send us an e-mail. We will take all feedback into account when releasing the next version of SIF.

6 Example Scenarios

SIF has been successfully used for the implementation of several application examples. In this section we will give a short description of each of them. All of the application examples have been conducted by students of the DFKI Multi-Agents Systems group.

6.1 The Automated Loading Dock

The Automated Loading Dock is a representative for many applications of agents in the manufacturing domain and has been designed in the MAS Group at DFKI as an evaluation scenario for broad agents, i.e., agents which have to behave reactively, deliberatively, and social at the same time. In the Loading Dock, forklift robots have to load and unload boxes of different categories from several docked trucks to corresponding shelves. Reactivity is required to implement the robust sensor-motor feedback of the robots for positioning, driving, and manipulating. Deliberative abilities are required to perform the delivery tasks and navigation that the forklifts equipped with. And social abilities are necessary in order to solve conflicts in the Loading Dock, e.g., by exchanging tasks and boxes, two robots can optimise their behaviour.

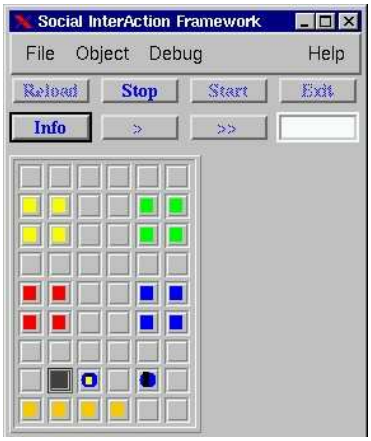

Figure 9 A screenshot of the loading dock simulation

The Loading Dock has first been implemented physically using Kephera robots controlled by the InteRRaP architecture (Müller 1996). In order to allow for controlled experiments, a simulation has been built using SIF. The special character of the Loading Dock as being a robotics domain is addressed by developing particular types of sensors and effectors.

6.2 Simulation of an Automated Trading system

Due to the great success of the new electronic media, in particular the Internet, new forms of communication, information retrieval and direct consumer commerce are beginning to arise. Electronic matching algorithms are being developed for the purpose of reducing market impact of large stock market transactions: The idea is that large institutional investors will be matched together, in an optimal sort of way, anonymously, so that markets do not have time to react to the buy/sell orders that would occur in traditional stock markets.

Using SIF, we have implemented simulations of some of those systems in order to gain experiences and to evaluate and compare the approaches. In our implementation we have realised a central matching engine and possible traders as SIF agents that communicate via the world server. Buyer and seller agents are equipped with the following effectors and sensors:

- *Logging and dis-logging effectors*: trader agents can register to a matching agent at any time. A bidder agent can also disconnect from a matching agent whenever the agent decides to do so.

- *Satisfaction transmission effectors*: a trader has to announce its satisfaction structure The. The satisfaction announcement has to be done right after the logging procedure but can also be performed during the run of the trading process if an agent changes its mind.
- *Information sensor*: once a deal is confirmed, confirmation messages sent by the matching agent are received in these sensors.

Trader agents can log in to different matching engines simultaneously if the traders wish to trade on different goods at the same time. The realised system is able to cope with the use of multiple matching agents. Agents of that type have the following sensors and effectors at their disposal:

- *Information effector*: if a matching agent has determined a seller-buyer pair for a certain trade it sends a message to both agents containing information on the price and quantity of the arranged deal.
- *Logging sensor*: using this sensor, a matching agent realises registration acts of trader agents.
- *Satisfaction transmission* sensor: a matching agent gets informed on satisfaction structure changes via this sensor.

The satisfaction functions or matrices of traders can be inserted conveniently in a script to be read during the start-up of the agents in SIF. Matching agents store all combinations of mutual satisfaction values of trading agents currently participating in a trading procedure in a priority queue, ordered by the value of the mutual satisfaction. For performance reasons this priority queue is realised by using a balanced (2, 4)-tree.

6.3 Resource Distribution Agents

Inspired by the strategic games which deal with building of settlements on a pre-defined map with a limited amount of resources and competing clans (the different players), agents have been designed which distribute a variety of different tangible resources.

In the application, each resource is controlled by an agent. These resource agents do not occur on the grid, but appear only as so called *virtual agents*. These virtual agents have sensors to perceive the world and specific effectors for manipulating the world, that is, distributing resources on the grid. Virtual agents do not appear on the grid and they have no representation which is perceivable by other agents. Due to this nature, limitations through restricted perception or action, which depend on the agent´s location on the grid are not applicable to virtual agents.

Resource agents can distribute resources on the grid. They can deposit the resources on a field or create resource sources, which provide unlimited resources of their kind. The distribution of resources can be done according to different distribution models, which can e.g. simulate the four seasons.

6.4 XSif – Search Agents

Different search strategies for agents in grid worlds have been implemented. With the helps of such a strategy an agent is able to systematically search the grid world for a specific target. Such a target can be another agent or special object (e.g. a block of a certain colour). Search agents have limited perception, their vision range is usually restricted to only a small fraction of the grid. In order to facilitate the search, the agent disposes of a GPS-sensor, which always tells it its own location on the grid. Implemented search strategies comprise a simple row search without a memory of searched areas on the grid. Another, more sophisticated

strategy is the map based search, where the agent builds a representation of the world and notes already searched spots on a map. This map is used by the agent to check for still undiscovered areas on the grid. Figure 10 shows an example of such a map right after the agent is created and after some exploration. It can be also displayed to the user to demonstrate the search.

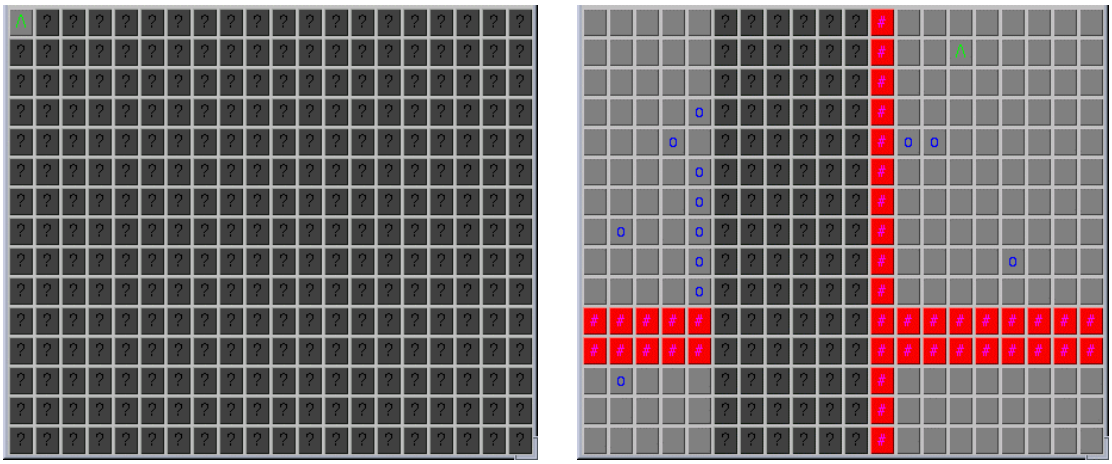

Figure 10 The map of the agent when started (left) and after some exploration (right).

6.5 The Pursuit Domain in SIF

Basic elements for standard predator prey scenarios have been implemented, better known as *pursuit games*. Located on the grid there is a prey agent and some predator agents (the actual number of predator agents varies). The goal of the predators is to come to a predefined capture position (e.g. to surround the prey, which means that each of its adjacent fields is occupied by a predator). The predators act independently from each other, they are not guided by some kind of central control unit. Nevertheless, in some scenarios they are able to communicate and cooperate.

In a basic scenario, the prey is known to the predators (an vice versa) and all agents have unlimited vision range. The basic capture position is defined as a position, in which the prey agent has no possibility to move either left, right, above or below.

The game parameters can be varied in a script file or in the implementation of an agent's strategy and in the implementation of the capture position. As an agent's strategy a user may choose one element out of a set of predefined strategies. A strategy defines an agent's reactions on sensorial or communicative stimuli. It also includes properties of its owning agent, which are directly related to the way, the agent is supposed to act.

7 Discussion

7.1 Conclusion

We have presented an architecture for building simulations, EMS, and its implementation, SIF, as a broad and reasonable simulation framework for multi-agent systems. The key design aspects have been the fine-grained virtually acting and sensing, the ability of rapid-prototyping simulated worlds, a broad implementation platform, and the possibility of supervising agents by human users. The testbeds we are currently developing with our tool, for example the automated loading dock, confirm the applicability of our approach. It has

been suggested that this framework is able to be the testbed for societies with emotional agents (Burt 1998).

7.2 Outlook

New developments in Virtual Reality (VR) have brought a qualitative change to human-computer interaction, in the form of co-habited virtual worlds (CHVW). In such worlds, synthetic agents and avatars (agents that are controlled and supervised by human users) interact in a globally networked setting. Applications of CHVW are, for example, virtual conferences where lifelikeness and interaction of avatars is a key issue. One could also think of virtual marketplaces incorporating electronic salesmen agents and customer avatars as a platform for future e-commerce. Similarly, the entertainment issue of interactive, virtual theatre requires mixed populations of synthetic characters and half-human, half-computationally steered personae. DAI technology; such as integrated into SIF; provides the key to develop the computational means of realising CHVW.

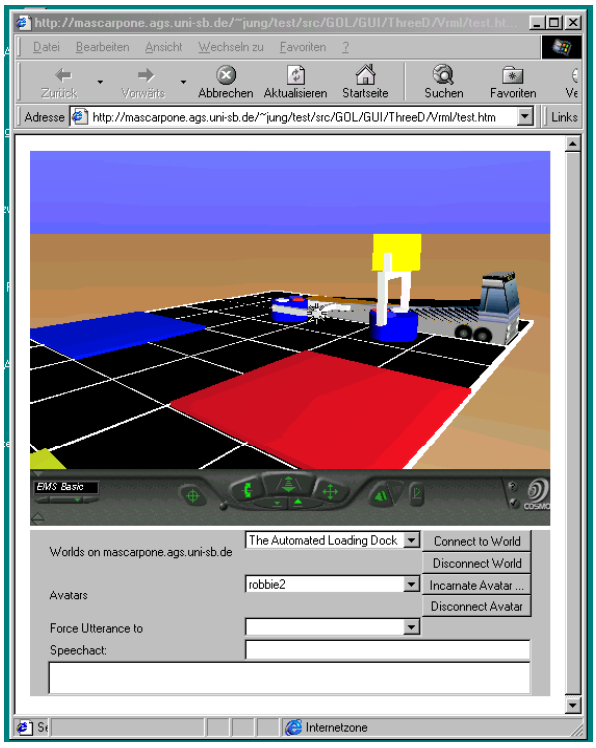

Figure 11 A screenshot of the upcoming SIF version

Indeed, the role of a human agent within SIF has already been addressed by the Control Window and the Control Pad GUI. The human perceives the world through a special sensor (his browser) and acts upon the world through a special effector (the control pad). Because of the restricted bandwidth of the human-computer interface; it is convenient to let particular agents (avatars) represent the human in the simulation. The agent perceives the world on behalf of the human and acts upon that perception accordingly. Thereby; the user is able to trace the avatar's behaviour and to guide and command its avatar through his action.

We are currently experimenting with a Web-Based User Interface to SIF which connects multiple human users remotely to a SIF simulation. It uses standard platforms; such as VRML, JAVA™; and RMI to visualise the state of the simulation and its change via three-dimensional; interactive graphics and animations. The key to do so is to apply asynchronous network technology. Asynchronous visualisation lacks the guarantee of persistence; but

enables to decouple the simulation from the computational operations of the clients. This is important for upholding the reproducibility of simulation results.

Network technology is the key to bring together various users from remote places and on different platforms. These advantages could be also transferred to SIF agents as well in order to move towards a distributed simulation: Agents and avatars could run on the machines of their users being under their control and not consuming central computational resources. Perception and action interfaces should be implemented as standard RMI or CORBA services being independent of any implementation language. And visualisation and appearance data can then be stored decentrally near the agent code and accessible to the global CHVW in form of URL's.

Acknowledgements

We would like to thank all the students who used SIF in their practice semester for their patience with revisions and their valuable feedback on the system. We would also like to thank Jasmin Schneider, our graphics designer, for putting our ideas into meaningful illustrations.

Bibliography

(Bouron et al. 1991) T. Bouron, J. Ferber, and F. Samuel. MAGES: A Multi-Agent Testbed for Heterogeneous Agents. In Y. Demazeau and J.P. Müller (eds.): Decentralised AI, volume 2, pp 195–216. North-Holland, 1991.

(Burt 1998) Burt, Alastair. Modelling motivational behaviour in intelligent agents in virtual worlds. In C. Landauer and K. L. Bellman (eds.), Proceedings of Virtual Worlds and Simulation Conference (VWSIM'98), Simulation Series vol. 2, The Society for Computer Simulation International, 1998.

(Funk et al. 1998) P. Funk, C. Gerber, J. Lind and M. Schillo. SIF: An Agent-Based Simulation ToolBox using the EMS Paradigm. In: *Proceedings of the 3rd International Congress of the Federation of EUROpean SIMulation Societies (EuroSim)*, 1998.

(Greenberg and Westbrook 1990) Michael Greenberg and David Westbrook. The Phoenix Testbed. Technical Report COINS TR 90-19, Computer and Information Science, University of Massachusetts, 1990.

(Hägg et al. 1994) S. Hägg, F. Yegg, R. Gustavsson, and H.Ottosson. DA-SoC: A testbed for modelling distributed automation applications using agent-oriented programming. I Proccedings of the Siyth European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-94), pp 39–51, August 1994.

(Hanks et al. 1993) Steve Hanks, Martha Pollack, and Paul Cohen. Benchmarks, Testbeds Controlled Experimentation, and the Design of Agent Architectures. AI Magazine, 14 (4), pp 17–42, 1993.

(Hart and Cohen 1990) David Hart and Paul Cohen. Phoenix: A Testbed for shared Planning Research. Proceedings of the NASA/AMES Workshop on Benchmarks and Metrics, June 1990.

(Jung et al. 1999) Christoph G. Jung, Jürgen Lind, Christian Gerber, Michael Schillo, Petra Funk and Alastair Burt. An Architecture for Co-Habited Virtual Worlds. In C. Landauer and K. L. Bellman (eds.), *Proceedings of Virtual Worlds and Simulation Conference (VWSIM'99)*, Simulation Series vol. 2, The Society for Computer Simulation International, 1999

(Khepara 1993) *Khepara Users Manual*. Laboratoire de Microinformatique, Lausanne 1993.

(Lind 1998) Lind, Jürgen. *The EMS model*. DFKI Technical Memo TM-98-09, December 1998

(Montgomery et al. 1992) T. Montgomery, J. Lee, D. Musliner, E. Durfee, D. Darmouth, and Y. So. MICE users guide. Technical Report CSE-TR-64-90, Department of Electrical Engineering and Computer Science, University of Michigan, 1992.

(Müller 1996) Müller, J. P. *The Design of Intelligent Agents: A Layered Approach.* Springer-Verlag, Lecture Notes in Artificial Intelligence, 1996

(Pollack and Ringuette 1990) Martha Pollack and Marc Ringuette. Introducing the tile-world: Experimentally evaluating agent architectures. In Proceedings of the Conference of the American Association for Artificial Intelligence, pp 183–189, 1990.

(Russell and Norvig 1996) Russell, S. and Norvig, P. A*rtificial Intelligence, A Modern Approach.* Prentice-Hall, 1996.

(Shoham 1991) Shoham, Y. Agent-oriented programming. In *Proceedings of the 11th International Workshop on DAI*, pages 345-353, 1991.

(Wiering and Dorigo 1998) Marco Wiering and Morco Dorigo. Learning to control Forrest Fires. In Proceedings of the Twelfth International Sysmposium on Computer Science for Environmental Protection, 1998.

(Wooldridge 1995) Michael Wooldridge. This is MYWORLD: The Logic of an Agent-Oriented DAI Testbed. In M.Wooldridge und N.R. Jennings (eds.): Intelligent Agents: Proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages. LNAI Springer-Verlag, 1995.

Abbreviations used

AOE	Abstract Object Engine
API	Application Programming Interface
EMS	Effector-Medium-Sensor (model)
GOL	Generic Object Library
GUI	Graphical User Interface
MAS	Multi-agent system
ROL	Runnable Object Library
SIF	Social Interaction Framework
VRML	Virtual Reality Mark-up Language

Appendix A The Packages

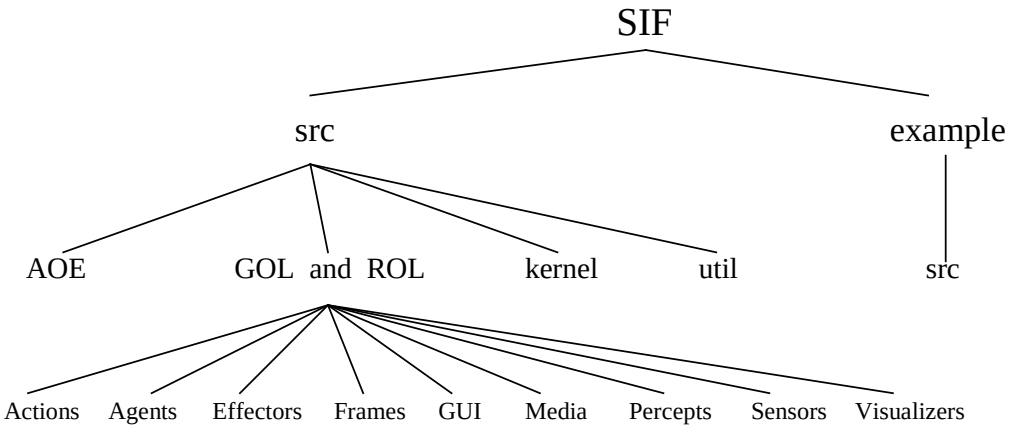

Figure 12 The directory structure of the SIF library

The source is distributed across a number of packages (see Figure 12). This distribution tries to reflect the EMS-paradigm. The *Abstract Object Engine* (AOE) contains the abstract definition of the classes that everything else inherits from. The *Runnable Object Library* (ROL) consists of ready-to-use objects. Any classes in between these two layers are in the *Generic Object Library* (GOL). Both the GOL and the ROL have the same directory structure (which in JAVA™ is the same as the package structure): they have domains for actions, agents, effectors, frames, GUI, objects, media, percepts, sensors and visualisers. In the kernel directory lie the core classes of SIF and util contains some auxiliary objects.

For designing your own simulation you will find the example directory most interesting as you can use it for cut-and-paste-programming. Its `src` directory has the same structure as the GOL and the ROL. Please note that the JAVA™ package names are created by concatenating the directory names while traversing the tree and including a „.“ in between two directory names.

Package: `example.src`

The classes that demonstrate a simple scenario for SIF. It can be used for cut and paste programming.

Package: `src.AOE`

The *Abstract Object Engine* (AOE) contains the basic classes from which others inherit, e. g. `src.AOE.Sensor`, `src.AOE.Medium`. Sometimes you also identify a class to belong to the AOE because it has the prefix „SIF“ to its name.

Package: `src.GOL.{actions, agents, effectors, frames, objects, media, percepts, sensors, visualisers}`

They are all part of the *Generic Object Library* (GOL). The GOL objects are independent of scenarios. They may be abstract classes that are implemented by objects in the next set of packages.

Package: src.GOL.GUI

The classes that deal with the *graphical user interface*(GUI), see Section 4.

Package: src.ROL.{actions, agents, effectors, frames, GUI, objects, media, percepts, sensors, visualisers}

They are all part of the *Runnable Object Library* (GOL). The ROL objects are implementations for special scenarios can be directly used.

Package: src.kernel

The core of SIF.

Package: src.util

A collection of auxiliary classes that are used by the SIF core and may be useful for other applications.

Appendix B Selected Classes

To conduct experiments you will most likely want to create a new scenario. In this section we describe which classes have to be overridden by a scenario specific implementation. They are all cited by their class name. To find the file `src.AOE.Medium` look for the file „Medium.java“ in the directory „src.AOE“. That means you append „.java“ to the last word in the class name. Figure 13 shows the inheritance structure of all classes in SIF and also their dependencies in the sense of which class uses which other class. Documentation of the fields and methods of these classes and more information can be found in the application programming interface (API) that can viewed online or downloaded from the homepage of SIF at DFKI.

File: `src.kernel.WorldServer`

Reference Implementation: `example.src.OurWorld`

This class contains the `main()` method which gets called from the JAVA™ virtual machine when you run SIF. It initialises all objects and passes their instances to the SIF core.

File: `src.AOE.Medium`

Reference Implementation: `src.ROL.media.*`

As mentioned above, these classes describe how the input for all sensors is to be computed. It also defines the effect of the actions on the world representation. We would like to stress the fact that these classes define operationally all effects of any action in the simulation. See the html documentation for the various class methods

File: `src.AOE.SIFGUIAgent`

Reference Implementation: `src.ROL.GUI.GUIAgent`

This is the heart of the *Graphical User Interface*(GUI). Here, the environment for the visualisation of a simulation is chosen (JAVA™, VRML™). The class provides a range of useful methods for the user interface apart from providing the mechanism to link the GUI with the SIF core.

File: `src.AOE.SIFCanvas`

Reference Implementation: `src.ROL.GUI.Canvas`

You need to change this only if you decide to give your agents or the simulated world a different appearance. Define here how the visualise all the objects that might appear in your simulations, depending on the contents of a data structure that is derived from `src.AOE.SIFFrame`.

File: `src.AOE.SIFObjectInfoGUI`

Reference Implementation: example.src.ObjectInfoGUI

This part of the GUI allows the user to check the internal state of an object (or agent) during runtime. This is a very powerful tool to keep control of the states of the objects while you run tests during the development of your simulation. With this free to edit GUI you need no longer massive text output to your shell. See also the html documentation on the visualiser objects that are available for a number of sensors. You can configure this GUI to output only the important data that is contained in your objects.

File: `src.AOE.SIFControlPad`

Reference Implementation: `src.ROL.GUI.ControlPad`

This class is the counterpart of the agent GUI. It allows the user to give an agent input, especially to make it perform specified actions, by clicking on buttons representing the actions an agent is capable of. By sending its data to the `GridAgentControlMedium` the command gets send to the agent the control pad is connected to.

File: `src.AOE.SIFWorldRepresentation`

Reference Implementation: example.src.WorldRepresentation

Depending on the scenario, a data structure to represent the state of the world has to be designed. This data structure must provide methods to access this representation and to update it.

File: `src.AOE.SIFIncorporateScript`

Reference Implementation: example.src.IncorporateScript

As SIF enables the user to run scripts, the script language has to be specified according to the needs of the scenario. Implementing the methods of this class provides for the specifications necessary. These will then be used when a script is loaded to initialise the representation of the world and the simulation.

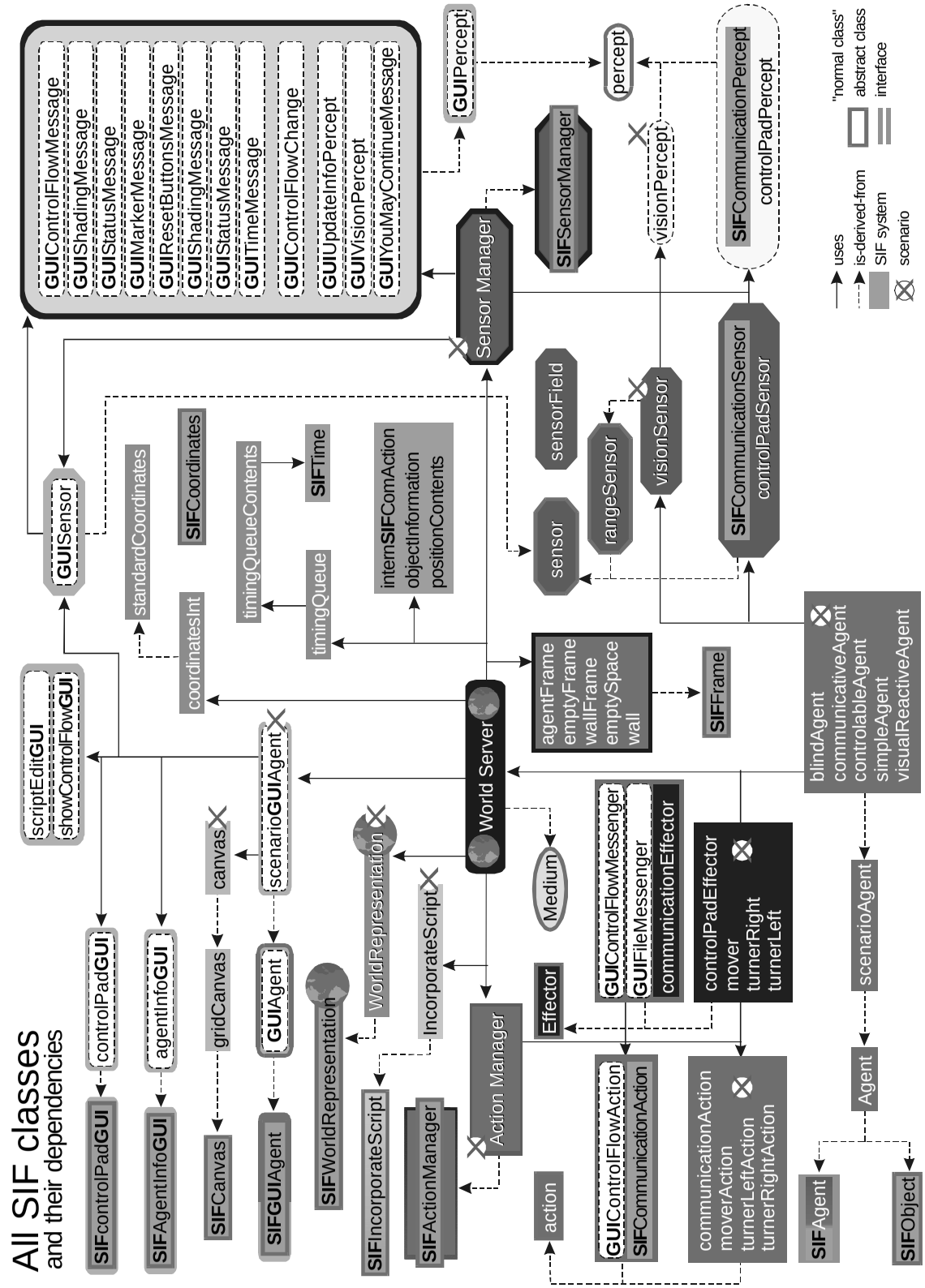

Figure 13 All classes of SIF and their inheritance and usage structure

SIF - The Social Interaction Framework

System Description and User's Guide to a Multi-Agent System Testbed

RR-99-02

Research Report

Michael Schillo, Jürgen Lind, Petra Funk, Christian Gerber

and Christoph Jung