An Executable Activity Theory Based Framework for Early Requirements Analysis

Rubén Fuentes
ruben@fdi.ucm.es

Jorge J. Gomez-Sanz
jjgomez@sip.ucm.es

Eva Ullán
evah@sip.ucm.es

Facultad de Informática
Universidad Complutense de Madrid
28040 Madrid, Spain

ABSTRACT

Gathering requirements in a domain problem is a challenging task for which several agent-oriented solutions have been devised. The high-level abstraction of agent concepts and their associated semantics eases this elicitation. Nevertheless, this does not ensure that the specification reflects what the customer demands. A frequent validation technique consists in building a prototype so that the customer can appreciate the result of the conversations with the analysts. The creation of such system requires time and money, and thus the industry would appreciate means of reducing these costs. This paper addresses this problem trying to prevent unnecessary developments. The proposal consists in capturing requirements with a framework based on the analysis of human societies named SCAT. This framework is generic enough to be domain independent and uses concepts similar to those in Agent-Oriented Software Engineering. Specifications captured with SCAT are described at low cost, since only general principles are required, and get the extra advantages of being both executable and verifiable. Besides, the resulting instantiation of the framework can be translated later onto a specific methodology if the required transformation rules are developed.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*elicitation methods, languages, tools*

General Terms

Documentation, Languages, Verification

Keywords

Social properties, Simulation, Formal proof, Activity Theory, Requirements elicitation, Agent-Oriented Software Engineering

1. INTRODUCTION

Defining valid and correct requirement specifications remains a challenging task for software projects. Given a domain problem, a development team may realize too late that the original requirements were wrong since they did not capture what the customer wanted. Any delay in this detection have an important impact in the project, as the later the mistake is found, the greater is the cost of fixing it. The application of agent concepts has provided some aids in the early detection of mistakes in these specifications.

The goal-oriented modelling promotes goals as the key concept in requirements. Lamsweerde reviews some of its approaches in [20], being two relevant examples Tropos [10] and KAOS [4]. Despite of their focus on goals, both of them require producing detailed specifications of the problem in order to evaluate the validity of the requirements. These extended specifications are far beyond the mere use of goals and they are described with some kind of formalism, whose use requires a high-level of expertise. Although either KAOS or Tropos could generate a prototype in the traditional sense, they provide also other alternative means to check the requirements without formal methods. These alternatives demand less effort to produce specifications than the formal options, but they are also less powerful analytical tools.

Non goal-oriented methodologies, like Passi [3] or ADELFE [1], deal with requirements by means of use cases. The way to proceed once use cases have been identified differs from one methodology to another, but all of them require a quite complete multi-agent system specification to be able to validate the requirements by execution. That is, these approaches manually validate the requirements using the prototypes.

Both goal and use case oriented approaches can benefit of automated code generation facilities. Examples of these processes can be found in Executable UML (Executable Unified Modelling Language) [13] and INGENIAS [11]. Executable UML (xUML) is an extension of UML 2.0 intended for model driven development with transformations. As xUML is object-oriented, it must be extended and adapted for its with agent concepts. This is the case of the work with protocols in Agent UML reported in [7]. INGENIAS also permits to quickly develop a system from a partially completed specification. In this case, the departure modelling language is agent-oriented and the prototypes are built by instantiation of code templates with information from models. This kind of model-driven developments reduces the cost of later modifications in the requirements, as an important part of the implementation is automatically made. Nevertheless, it is subject of the same restrictions of the previous approaches, as there are no specialized means of

Jung, Michel, Ricci & Petta (eds.): *AT2AI-6 Working Notes, From Agent Theory to Agent Implementation, 6th Int. Workshop,* May 13, 2008, AAMAS 2008, Estoril, Portugal, EU.

capturing and validating requirements others than carrying out an agent-oriented development.

Related with both the formal analysis and the use of prototypes, this paper proposes validating a requirement specification by means of its execution, the inspection of its traces, and the verification of developer-defined properties. Proposing the direct execution of specifications to ensure their validity is not new. For instance, Gravell and Henderson [12] defend the benefits of executable specifications and recommend complementing them with methods like manual or automatic inspection of the specifications. In order to be useful in this setting, the effort to produce these specifications should imply a lower cost than producing a prototype or a formal specification. Note that this line is different from that of code generation. The xUML or INGENIAS models are transformed to generate prototypes that are the object of the verification while, in this case, checking happens over the original specifications.

The approach in this paper addresses requirement specification with the SCAT (Situation Calculus for Activity Theory) framework for the development of Artificial Societies based on the Activity Theory. The Activity Theory (AT) [21] is a paradigm from the Social Sciences that analyzes human societies focusing on their contextualized activities. An Artificial Society (AS) [18] is a synthetic representation of a society. AS are related with the computational study of societies but not particularly with AT. The operational semantics of SCAT is an AT extension of the Situation Calculus according to the ConGolog [5] implementation. The advantage of this formalism is that its implementation enables executable specifications. A SCAT specification uses high-level AT concepts with semantics close to agent-oriented ones [8]. This allows transferring [9] the acquired knowledge to an agent-oriented methodology, which is better suited for the engineering of the system. When compared with other alternatives, SCAT offers some features that suggest it can trim down the cost of checking requirements through execution:

- SCAT handles only concepts of high level of abstraction. Hence, a SCAT specification is not blurred by low level details.
- The number of elements needed to describe a meaningful specification is low. It suffices declaring the participating entities, dependencies, and what is going to be run to perform validation tests.
- SCAT predicates semantics are intuitive. They are based on common concepts of everyday life used by AT to explain the behaviour of human societies.
- SCAT does not compel to incorporate time reasoning in the specification. This prevents effort on behalf the developer.

These advantages imply in exchange certain tradeoffs in the capabilities of SCAT. The most relevant one is that the level of detail that SCAT can capture is reduced if compared to what other formalisms [4, 10] allow. However, it must be noticed that SCAT is not intended to capture formally the requirements, but to provide an inexpensive way of early detecting mistakes in the specification, so that the real development can start in better conditions.

The rest of the paper is structured as follows. Section 2 briefly presents AT. The SCAT framework is introduced in Sect. 3, which describes the language and its grammar, and Sect. 4, which reports its implementation using ConGolog [5] and some hints about its execution. Section 5 shows an example of requirements elicitation and specification with SCAT. Section 6 describes how a SCAT program can be translated to a software methodology, INGENIAS [11] in this case. INGENIAS was chosen due to its extensive tool support for visual modelling, code generation, and reporting facilities. Related work regarding the execution of specifications as means to increase the knowledge about requirements is discussed in Sect. 7. Finally, we present some conclusions about the approach.

2. ACTIVITY THEORY

The Activity Theory (AT) [21] is a framework for the analysis of human groups focused on their contextualized acts. These acts are called *activities* and constitute the minimal meaningful unit to understand human actions. Their context comprehends the socio-physical environment and their historical development. This section introduces the main AT concepts through examples from the case study of this paper.

The term activity [6] refers to a process intended to transform some object into the outcome that satisfies the needs of the subject of that activity. For instance, a researcher working in a department do research transforming the available knowledge in papers that contain the solution for a given problem. The subject is the participant or group whose agency is chosen as the point of view in the analysis. Potential subjects are the agents, people using a software system, or their societies. In our example, the researcher is the subject. The objects of the activity can be raw materials or mind structures. Our researcher transforms knowledge to generate new knowledge for a paper. The subject's needs that motivate the performance of the activity are their objectives. Some objectives for our researcher can be contributing to mankind knowledge or obtaining merits for a job post.

The relations between subject and object are not direct but mediated by several kinds of elements. The subject always acts over the object through tools. Tools can be physical or symbolic, external or internal, and crystallizes the experience of the group in the process. Following our example, the researcher uses as tools previous experiences writing papers and devices like the laptop or the word processor. The community includes those subjects related in any way (e.g. by use, ownership, or awareness of the existence) with the objects of the activity. Communities in the research activity can be that of the researchers in the same field, the department where the researcher works, or the family and friends, as all of them may have some kind of influence over the activity. A community is moulded by the division of labour and the rules. The division of labour establishes the role of the actors in the community, the power that they hold, or the tasks they are responsible of as related with the transformation process. The rules, norms, and conventions of the communities that constrain the activity are encompassed under the concept rules. These mediating entities and their relations make explicit the influence of the environment in which the activity happens and the historical and social development of it.

All the previous elements (i.e. an activity and its subjects, objects, outcomes, tools, objectives, communities, rules, and divisions of labour) constitute an activity system, that is,

Program	=	Neighbourhood$^+$ Entities$^+$ Relations$^+$ ATDeclaration* Instance$^+$ Universe$^+$
Neighbourhood	=	**neighbourhood**(*ID*, [IDList]).
Entities	=	**entities**(*ID*, IDList, [ElemList]).
ElemList	=	**elem**(*ID*, ATRole) | **elem**(*ID*, ATRole), ElemList
ATRole	=	**activity** | **subject** | **object** | **tool** | **outcome** | **objective** | **community** | **rules** | **divisionOfLabour** | **action** | **operation**
Relations	=	**relations**(*ID*, [IDList], [RelList]).
RelList	=	**rel**(*ID*, ATRel, *ID*, *ID*) | **rel**(*ID*, ATRel, *ID*, *ID*), RelList
ATRel	=	**executedBy** | **transform** | **use** | **produce** | **try** | **accomplishedBy** | **ruledBy** | **organizedBy**
Instance	=	**instance**(*ID*, *ID*, [IDList]).
ATDeclaration	=	**pursue**(*ID*, *ID*). | **decompose**(*ID*, [IDList]). | **duration**(*ID*, *Int*).
Universe	=	**universe**(*ID*, [IDList], [IDList]).
IDList	=	*ID* | *ID*, IDList

Figure 1: SCAT grammar

the context of the activity. Such a context is a mandatory prerequisite to understand truly the whole system and its ruling principles, and to improve its engineering [6].

At the same time, an activity system is made up of, and embedded into, neighbourhoods of nested activities, actions, and operations, all of which could be conceived as separate activity systems depending on the observer's perspective. An action is a task that pursues goals non-relevant individually but that contribute to a primary goal. For instance, writing the paper is part of the research and, although required to complete the activity, does not satisfy by itself the objective of generating knowledge. An operation is a concrete realization of an action for specific circumstances. For instance, writing the paper can be done in a word processor or by hand. In these neighbourhoods, factors that affect the current activity system could be the outcome of other activity systems. Typical examples of these situations are activities that produce the components of the activity systems of other activities or even new activities. For instance, the researcher produce some experimental results that are part of the knowledge transformed as objects in the writing of the paper. For the remaining discussion, the term task stands for activities, actions, and operations when the differences between them are not relevant. The term artifact refers to any concept of an activity system.

3. SCAT GRAMMAR AND INFORMAL SEMANTICS

SCAT (Situation Calculus for Activity Theory) is a modelling, simulation, and verification framework for the analysis of societies. It is intended for practitioners of Social Sciences through a domain specific language [19]. The language allows specifying AS following the AT guidelines seen in Sect. 2. SCAT specifies AS with universes made up of neighbourhoods describing general patterns of behaviour and instances of them that characterize actual behaviours in the system.

Figure 1 presents the main rules of the grammar of SCAT with an EBNF-like notation. Terminal symbols are highlighted using a bold font. All the identifiers of elements are unique in the specification, although an element (but not a relation) can be declared several times with different roles. For instance, the same element can be the outcome of an activity and be used as a tool in other activity. Notice that following the Prolog notation used in ConGolog [5], square brackets do not indicate optional elements but are the terminal symbols for lists. For instance, *[ElemList]* corresponds to a SCAT list where *ElemList* defines its elements separated by commas.

A SCAT program comprehends declarations about several aspects, being some of them optional. A *neighbourhood* defines a parameterized network of activity systems with its participants declared in the *entities* declaration and their dependencies in the *relations* declaration. *ATDeclarations* are primitives adding extra information about individual entities. The *instances* define the actual activity systems by grounding some of the neighbourhoods. The *universe* is the set of instances that are going to be studied along with the actual participants that exists previously to the execution of any activity.

A *neighbourhood* declaration associates an identifier *ID* to a network of linked activity systems. It allows declaring as parameters the entities for the neighbourhood that can be externally set with the argument *IDList*. This way, the neighbourhood declaration can be reused with different actors.

The entities in the neighbourhood need to be associated to the roles they play within its activity systems. The *entities* declaration has as arguments, in order of appearance, the *ID* of the neighbourhood previously defined, a list of parameters that is the same as in the corresponding neighbourhood, and a list of elements. Each *elem* description includes an *ID* for its element, which can match a particular entity of the world or one of the parameters, and the role that it plays in that neighbourhood according to the AT. The different roles are extracted from the AT formalization presented in [8]. They correspond to the entities used to depict an UML-AT diagram. For more information, we invite readers to review the original work.

The *relations* declaration describes the dependencies between entities according to the AT. The first two arguments are the same as for an *entities* declaration. The last argument is the list of relationships among the entities previously defined for the neighbourhood. A *rel* declaration has four arguments: an *ID* for the relationship, the kind of AT relationship, and the entities it connects. The different kinds of relations are extracted as well from [8]. According to that definition, the elements connected by the relationship must play concrete AT roles. For instance, an **executedBy** rela-

tion indicates that a subject is responsible of the execution of an activity. Hence, it declares a capability of the subject. Another example is **produce** that establishes the concrete outcome generated by the execution of an activity.

An *instance* declaration establishes a parameterization of a neighbourhood. Its first two arguments are the identifiers of the instance and the associated neighbourhood, and the third one is a list of values that grounds the parameters of the neighbourhood.

Besides the general declarations about activity systems, SCAT allows making statements about individual elements with *ATDeclarations*. The declaration **pursue** establishes that a subject tries to achieve an objective. A declaration **decompose** describes how an artifact is obtained composing other artifacts (this is the case for activities realized through actions and actions through operations). Finally, **duration** expresses how many units of time take the execution of an operation. Without an explicit declaration, an operation needs one unit of time.

Finally, a *universe* refers to a set of instances whose evolution along time has to be observed. They correspond to scenarios where the different actors are tested. The arguments of this declaration are the identifier of the universe, since there can be several under inspection, its list of instance identifiers, and a list of elements initially available. All the elements existing in the universe must appear in this declaration or be produced by some activity.

4. SCAT IMPLEMENTATION

The predicates in the previous section characterize universes independently of their execution model. This allows certain types of static verification of the systems (like finding contradictions in the specifications in the line of [8]), but it does not allow checking their dynamic behaviour. For this purpose, SCAT needs an operational semantics that determines how universes evolve over time with the execution of tasks and the way in which subjects select the tasks to execute. This evolution has two important requisites. First, interruption of tasks must be allowed at any moment, as subjects can be unable, or do not want, to finish them. Second and to ease the analysis, the semantics must provide a sorted account of the events in the execution over a timescale. The issue of the universe evolution is addressed through an extension of ConGolog [5]; a standard implementation of the choice of tasks is provided following the Rationality Principle [15].

The introduction already points out that the operational semantics of SCAT is built upon ConGolog, which is a Prolog implementation of the Situation Calculus. The Situation Calculus (SC) deals with the evolution of a system, whose initial state is a term `s0`, by means of actions performed over states (the ConGolog term `do(action, state)`). The pre and post conditions of these actions are determined through customized frame axioms. These axioms determine what changes as a consequence of the execution of the actions and also explicitly state what remains unaltered after an action execution.

There are other extensions of SC different from ConGolog. Concretely, Pinto and Reiter [16] or Zimmerbraun and Scherl [22] deal with the integration of time into SC, but the results do not meet the needs of this concrete development. Pinto and Reiter [16] do not permit interruptions in the execution of activities and Zimmerbraun and Scherl [22] do not indicate when events appear over an absolute timescale. The solution applied in SCAT uses flags similar to those of [22] to mark the beginning and the end of the different tasks, as well as absolute marks of time in the different states, similar to those of the Event Calculus [14].

SCAT over ConGolog

Roughly, a universe is regarded as a set of ConGolog systems where multiple subjects simultaneously perform tasks. These tasks affect to their activity systems and have collateral effects in other systems through shared elements. The tasks need a time for their execution whose account demands an explicit representation of time. For this purpose, SCAT codifies its own quanta of time. Every universe includes a special non-declared subject called `cronos` that is responsible of time passing. It executes an atomic activity called `tick` dividing the quanta. A basic operation may begin and end its execution at the same or different quanta depending on its duration. To point out these facts in the SCAT framework, the basic unit of operation becomes the `doing` term:

```
doing(Context, ID, Flag, Subject, Task, Params)
```

It corresponds to the begin or the end of the execution of a task (i.e., AT activity, action, or operation). Its parameters are identifiers of elements in a neighbourhood plus some additional elements for the management of the doings. A `Subject` executes the doing related with a `Task`. The `Flag` takes values among three terms: `shot`, `begin`, and `end`. The value `shot` is only used with environment events that take zero units of times (i.e., they happen in just one quantum); `begin` and `end` indicate the start or the end of the execution of a task respectively, and are used in all the other cases. Each task execution in a run has an identifier `ID`, that is unique for a doing term with a shot flag, but shared by the terms for the same execution of a task with flags begin and end.

Following the SC, the evolution of a universe is described as a sequence of doing terms from an initial state `s0`. With the exception of the subject cronos, whose choices of doings are part of the execution environment, all the other subjects are able to decide when they want to execute a doing. These choices are considered with the `chooseDoing` predicate:

```
chooseDoing(Subject, State, Doing)
```

The predicate takes the identifier of the subject that has to choose a doing at a given setting indicated by `State`. This state is a sequence of doings beginning in `s0`, or the special term `now` to indicate the current state as in ConGolog.

Analysis of Requirements

SCAT programs built upon ConGolog get benefit of the standard resolution algorithm of Prolog, although it affects its scalability. Concretely, they can return every possible result (i.e. sequence of doing terms) that satisfies a given universe. SCAT programs are executed looking for the equilibrium of the society or the possibility of reaching a state that satisfies some constraints. The concept of equilibrium refers to a state where some society features do not change or they change in cycles. Examples are those of a universe completed or one where a subject always performs the same activity without achieving the desired effect in the environment. The second group of tests refers to the existence of

neighbourhood(research,
[Researcher, Paper]).
entities(research, [Researcher, Paper],
[**elem**(Researcher, subject),
elem(write, **activity**),
elem(createKnowledge, **objective**),
elem(Paper, **outcome**)]).
relations(research, [Researcher, Paper],
[**rel**(r1, **try**, write, createKnowledge),
rel(r2, **executedBy**, write, Researcher),
rel(r3, **produce**, write, Paper)]).
duration(write, 2).

neighbourhood(teaching, [Teacher]).
entities(teaching, [Teacher],
[**elem**(Teacher, **subject**),
elem(instruct, **activity**),
elem(transmitKnowledge, **objective**)]).
relations(teach, [Teacher],
[**rel**(r4, **try**, instruct, transmitKnowledge),
rel(r5, **executedBy**, instruct, Teacher)]).

neighbourhood(job, [Employee, Post]).
entities(job, [Employee, Post],
[**elem**(Employee, **subject**),
elem(apply, **activity**),
elem(getJob, **objective**),
elem(Post, **object**),
elem(rule(unique, [Post]), **rules**)].
relations(job, [Employee, Post],
[**rel**(r6, **try**, apply, getJob),
rel(r7, **executedBy**, apply, Employee),
rel(r8, **transform**, apply, Post),
rel(r9, **ruledBy**, apply, unique)]).
rule(unique, [Post]) : −
constraint([], [instance(_, job, [_, Post])]).
rule(exist, [Employee, Post]) : −
constraint([instance(_, research, [Employee, _])],
[instance(_, job, [_, Post])]).
decompose(apply, [instruct, ask]).
decompose(apply, [instruct, write, ask]).

pursue(john, createKnowledge).
instance(disia1, research, [john, scatPaper]).
instance(disia2, research, [paul, scatPaper]).
pursue(john, transmitKnowledge).
pursue(paul, transmitKnowledge).
instance(disia3, teaching, [john]).
instance(disia4, teaching, [paul]).
pursue(john, getJob).
pursue(paul, getJob).
instance(fdi1, job, [john, lecturer]).
instance(fdi2, job, [paul, lecturer]).
universe(ucm1, [disia1, disia2],
[john, paul, scatPaper, createKnowledge]).
universe(ucm2,
[disia1, disia2, disia3, disia4, fdi1, fdi2],
[john, paul, scatPaper, lecturer,
createKnowledge, transmitKnowledge, getJob]).

Figure 2: SCAT program for the life in the department.

a given sequence of tasks in the universe that, if performed by their subjects, generates a state where some given properties holds. To perform the previous tests, SCAT provides the predicate `deriveSequence`:

```
deriveSequence(State, Sequence, PosConstraints,
                                NegConstraints)
```

`State` represents a specific initial setting of interest in the considered universe. `PosConstraints` is a list of AT entities (see Sect. 2) or instances of neighbourhoods that have to appear in the evolution of the system from the initial state to the final one. In the case of activities, they have to be executed and completed (satisfying or not their objectives); objectives have to be achieved; the other AT entities have to exist in `State` or being produced by some completed activity of the sequence; instances have to achieve all the objectives in their activity systems. `NegConstraints` is a list of elements that must not appear or be satisfied in the sequence. Last, `Sequence` is a sequence of doings that makes evolve the initial `State` to one that satisfies the constraints. A query about simulation ending just imposes empty lists of constraints, both positive and negative. An example of a query about the reachability of a state is:

```
deriveSequence(s0, Sequence, [elem(think, action),
          elem(explain, action),
          elem(scatPaper, outcome)], []).
```

The query tries to obtain a `Sequence` of doings from the initial state `s0` that leads to a state where the outcome `scatPaper` has been produced, and the actions `think` and `explain` have been completed.

5. EXAMPLES OF SCAT PROGRAMS: LIFE WITHIN THE ACADEMY

This section introduces part of a case study about life in a university whose goal is to construct a simulator for that setting. The current example focuses on how a person gains merits to obtain a job in a university department. There are two alternatives: to do research and teaching; or just teaching. Departments expect that all their applicants adopt the first option as it gives them a higher self-esteem and prestige to their departments. John and Paul are two applicants for a job. John has in his CV teaching and researching activities, and Paul has only teaching. The simulation refers to who will gain the position and how. Figure 2 shows the SCAT program with three neighbourhoods (for doing research, teaching, and applying to a job), and the instances and universes for their analysis.

The neighbourhood *research* explains how and why a researcher writes papers. Essentially, a *researcher* intends to *create knowledge* by an activity called *write* that produces a *Paper*. The activity write has a duration of two units of time. There is also a simpler neighbourhood for *teaching* with just one activity. The last neighbourhood explains how the job can be gained. An *employee* makes merits and then triggers the action *ask*. These actions make up the activity *apply*. The only applicant who complete the activity gets the job. The uniqueness is determined by the rule *unique*.

The two universes at the end of Fig. 2 represent potential scenarios for the system. The universe *ucm1* will tell who actually writes the paper when John and Paul are able to

?- deriveSequence(s0, Sequence, [elem(disia1, instance)], [elem(fdi1, instance)]).

Sequence = do(doing(nil, 12, end, paul, wait, [defining(wait, paul)]),
do(doing(nil, 12, begin, paul, wait, [defining(wait, paul)]),
do(doing(nil, 11, end, john, wait, [defining(wait, john)]),
do(doing(nil, 11, begin, john, wait, [defining(wait, john)]),
do(doing(nil, 10, shot, cronos, tick, [time(3)]),
do(doing(nil, 4, end, paul, apply, [defining(job, paul, lecturer)]),
do(doing(4, 7, end, paul, ask, [defining(job, paul, lecturer)]),
do(doing(2, 9, begin, john, ask, [defining(job, john, lecturer)]),
do(doing(nil, 1, end, john, write, [defining(research, john, scatPaper)]),
do(doing(nil, 8, shot, cronos, tick, [time(2)]),
do(doing(4, 7, begin, paul, ask, [defining(job, paul, lecturer)]),
do(doing(4, 5, end, paul, instruct, [defining(teaching, paul)]),
do(doing(2, 3, end, john, instruct, [defining(teching, john)]),
do(doing(nil, 6, shot, cronos, tick, [time(1)]),
do(doing(4, 5, begin, paul, instruct, [defining(teaching, paul)]),
do(doing(nil, 4, begin, paul, apply, [defining(job, paul, lecturer)]),
do(doing(2, 3, begin, john, instruct, [defining(teaching, john)]),
do(doing(nil, 2, begin, john, apply, [defining(job, john, lecturer)]),
do(doing(nil, 1, begin, john, write, [defining(research, john, scatPaper)]), s0)...)

Figure 3: Trace generated by the execution of the SCAT program

do it. That is, who will successfully complete one of the two instances *disia1* or *disia2*. The initial elements available at *ucm1* are the subjects, a paper to write, and the will of creating knowledge. The second universe, i.e., *ucm2*, mixes several instances where Paul and John write papers, give lectures, and pursue a job. It tests who gets the job as lecturer in the department by completing the instance *fdi1* or *fdi2*. The execution would start from the subjects, a paper to write, the post, the will of creating and transmitting knowledge, and the goal of getting a job.

Figure 3 includes a query about if it is possible that an applicant with research does not obtain the job in the department and a possible answer of the reasoning system. The activity systems use the default behaviour for the choice of doings. The fact that John made research corresponds to the satisfaction of the positive constraint about the instance *disia1*, while that John did not get the job is the negative constraint about the instance *fdi1*. Remember from Sect. 4 that a constraint about an instance is satisfied when all the objectives of its neighbourhoods are satisfied.

In this case, the system finds a solution for the query that also appears in Fig. 3. The subject Paul, who does not pursue the objective *createKnowledge* about doing research, is able to get the job, as one of the alternatives to achieve this goal just requires making teaching before asking the job. Paul begins

```
do(doing(4, 5, begin, paul, instruct,
   [defining(teaching, paul)])
```

in quantum 0 and finishes it in the next quantum, when he also begins the activity *ask*. In quantum 2, he gets the job as he satisfies all the requirements. As only one subject can obtain the job because of the applicable rule *unique* in the instance, the subject John, who teaches and does research, loses the job. The parameters just indicate the actual instantiation of the instances.

Another relevant issue is the flow of time. It appears as the doings of the activity tick carried out by the subject cronos. Finally, the execution is considered as finished when no subject (cronos excluded) can execute an activity different of wait, which does not produce any change in the universe. The subject Paul has satisfied his only objective with the activity *apply*. The subject John has made research but he cannot perform apply to satisfy the other objective because of the rule *unique*. Thus, both of them can only perform *wait* by quantum 4. This situation could be avoided substituting the rule *unique* by *exist*, which demands that the applicant do also research.

6. INTEGRATION WITH AGENT-ORIENTED METHODOLOGIES

Though executable, a SCAT program cannot be considered as a fully developed system. It is a resource to capture and execute activity systems at low cost, whose outcome is knowledge about the system to be. To have a complete running system, this knowledge must be integrated in the process of a software engineering methodology. Given that SCAT works on concepts from AT, choosing an agent-oriented methodology makes sense. The main reason is that the interpretation of intentionality embedded within SCAT resembles to the BDI [2] principles. SCAT considers in its default implementation, although it is not constrained to, subjects that accomplish activities in order to achieve objectives.

Applying the knowledge represented by a SCAT program in an agent-oriented methodology is done by producing a tentative system specification following the notation of the selected methodology. This problem has been already addressed by the authors in [8], where translations are provided from several methodologies to the AT based language UML-AT. Since SCAT borrows many concepts from UML-AT, it is natural to reuse those translation facilities.

Figure 4 shows some examples of the equivalence of concepts between AT and the agent-oriented methodology INGENIAS [11]. The different networks of AT concepts appear on the left and their correspondences in INGENIAS to the right. These AT concepts matches some of the ones pre-

AT	INGENIAS
Activity →/accomplishedBy/→subject →/transform/→object →/produce→outcome →/try/→objective	Agent Model: agent(subject) →/WFResponsible/→task(activity) →/GTPursues/→goal(objective) Tasks and Goals Model: task(activity) →/GTAffects/→fact(object) →/WFProduces/→fact(outcome) →/GTSatisfies/→goal(objective)
Community →/decompose/ →[subject1, subject2]	Organization Model: group(community)→/OHasMember/ →agent(subject1) →agent(subject2)

Figure 4: Some transformation rules from UML-AT to INGENIAS

sented when reviewing SCAT grammar in Sect. 3. These networks are represented as entities connected by relationships represented with arrows labelled within "/". In some cases, the same networks of elements can match several translating networks, both from AT to the agent-oriented methodology or vice versa. This ambiguity problem cannot be solved trivially. It requires human assistance to determine which translation is more appropriate in the current problem. To save some effort, for a concrete network of AT elements and agent-oriented methodology, the selection is made only once.

Following this mapping table, and using the developer intervention to disambiguate terms, the neighbourhood with identifier *job* shown in Fig. 2 can be translated to the INGENIAS diagram in Fig. 5. The transformation is not complete, since some elements from SCAT have no correspondence in INGENIAS, like the duration predicate. In this case, this predicate was translated as TextNotes entities, which are free text annotations included in the diagrams.

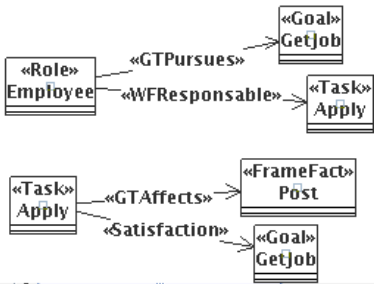

Figure 5: Partial transformation of the SCAT program

As expected, the size of the diagram is small since there is a high compatibility between BDI and AT concepts. To evaluate if the SCAT language really saves effort in the specification, we tried to make the program in Fig. 2 progressing towards a prototype with the INGENIAS Agent Framework (IAF). The first table, labelled as Result in Fig. 6, shows stats about the number of entities and relationships of the initial translation to INGENIAS. Notice that some original elements can be translated several times because different rules from the mapping provide different interpretations for them. This initial translation is not directly executable. It is a valid INGENIAS specification but its inconsistencies and the lack of certain elements make that the IAF cannot generate code for it. So a quick refactoring is made with the initial specification in order to remove all errors. In the initial translation to INGENIAS, the total amount of elements, entities or relationships, summed up to 28 elements. After making it IAF compatible, the specification size increased to 43 elements. Further application of the INGENIAS methodology - the definition of the deployment, external components for paper writing and application deliver, code associated to tasks - lead to 66 elements, an even greater increment. Given that INGENIAS has a quick implementation stage due to its code generation facilities, these figures and effort can be considered as a low limit to build a prototype for agent-oriented methodologies. It should be expected even wider differences with other methodologies less supported by tools.

Finally, remarking that though INGENIAS has been considered in this section, SCAT could be translated to other methodologies due to its roots in AT. Work made in agent model integration with AT in [9] justifies this statement.

7. RELATED WORK

The production of requirements specifications has been considered in the agent literature. This section focuses on two works highly related with early requirements specification, goal operationalization in KAOS and Formal Tropos.

Using the KAOS methodology, executing the requirements specification is studied as the operationalization of goals in [17]. This operationalization adds similar features to the work presented here with some remarks. A SCAT program is executable by itself. However, [17] requires defining additional information, like the preconditions and postconditions of the operations required to achieve the goals. On the other hand, the verification of properties in SCAT is made with a basic searching algorithm, whereas [17] bases on model checking tools. The animation of the execution is made graphically in [17] by means of goal state machines intended to express the operation descriptions. The research in this paper depends on the Prolog facilities to animate the specification. Therefore, the user can inspect the current evaluation of the different variables and the incorporation of new terms to clauses.

Formal Tropos [10] permits precise definitions of systems with time-based predicates about their behaviour. Model checking based tools are used to inspect this formal specifi-

Result	
Name	Times
Agent	2
FrameFact	2
GTAffects	1
GTPursues	1
GTSatisfies	2
Goal	2
Role	4
Task	4
TextNote	2
WFDecomposes	1
WFPlays	4
WFProduces	1
WFResponsable	2

Refactorized	
Name	Times
Agent	2
FrameFact	6
GTModifies	1
GTPursues	1
GTSatisfies	4
Goal	2
Role	4
Task	4
TextNote	2
WFConsumes	6
WFDecomposes	1
WFPlays	4
WFProduces	2
WFResponsable	4

Figure 6: Some transformation statistics for the overall universe

cation in order to verify the satisfaction of properties. On the other side, SCAT does not require including reasoning about time while specifying the system. Time is considered only during the execution. This makes a SCAT program simpler than a Formal Tropos specification. At the same time, it could be argued that Formal Tropos is more expressive than SCAT, at least in what refers to time aspects.

8. CONCLUSIONS

This paper has shown the application of SCAT, a framework for the analysis of activity systems, to early requirements gathering and validation. SCAT allows the quick specification of models of the systems under study. With little information, users are able to run simulations of the expected behaviour according to the AT, and checking hypothesis by means of their models.

Validation of requirements is achieved by checking the results obtained with simulations. These simulations can be used as well to determine if some states can be reached or not. It is assumed that the final validation happens as a result of confirming with customers the appropriateness of the simulation behaviour.

The SCAT specification can be later on translated to other methodologies for development. As an example, this paper has presented some results applied to the INGENIAS methodology. Nevertheless, SCAT can be translated to others, as proves some seminal work made in model integration with AT [9]. This translation has also been useful to show that SCAT specifications can provide simulation and verification with less effort then the required to build a prototype in an agent-oriented methodology, despite of the extensive tool support of INGENIAS.

The current model of SCAT has some limitations. The first one refers to its non-monotonic reasoning due to Prolog, which does not guarantee an answer to a query in finite time and hinders scalability of SCAT programs. To prevent this, model checking techniques are being studied. The idea is to transform a SCAT program into a suitable input for a model checking engine. Secondly, the specification of complex systems is still a tedious work subject to errors, as it needs many predicates. Here, we are working in the integration between SCAT and UML-AT modelling tools. Our purpose is to provide customized graphic modelling tools for SCAT. The last limitation emerges from the trace of the execution of universes. The state of a universe according to the SC is a chain of actions from an initial state. These traces become difficult to study very soon in the execution. New predicates will be added to SCAT to simplify the analysis of states in order to solve common queries. Besides, tools to animate specifications are being considered.

9. ACKNOWLEDGMENTS

This work has been funded by the Spanish Council for Science and Technology under grant TIN2005-08501-C03-01, the Dirección General de Universidades e Investigación de la Consejería de Educación of the Comunidad de Madrid, and the Universidad Complutense de Madrid (Research Group 921354).

10. REFERENCES

[1] C. Bernon, M. Gleizes, and G. Picard. Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology. *Agent-oriented Methodologies*, 2005.

[2] M. E. Bratman. *Intention, Plans, and Practical Reason*. CSLI Publications, 1987.

[3] M. Cossentino and I. ICAR-CNR. From Requirements to Code with the PASSI Methodology. *Agent-oriented Methodologies*, 2005.

[4] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Selected Papers of the Sixth International Workshop on Software Specification and Design table of contents*, pages 3–50, 1993.

[5] G. De Giacomo, Y. Lespérance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[6] Y. Engeström. *Learning by Expanding: an Activity-Theoretical Approach to Developmental Research*. Orientakonsultit, 1987.

[7] J. O. et al., editor. *Representing Agent Interaction Protocols with Agent UML*, volume 3382 of *LNCS*. Springer, 2005.

[8] R. Fuentes, J. J. Gómez-Sanz, and J. Pavón. Activity theory for the analysis and design of multi-agent

systems. In P. Giorgini, J. P. Müller, and J. Odell, editors, *AOSE*, volume 2935 of *LNCS*, pages 110–122. Springer, 2003.

[9] R. Fuentes-Fernández, J. J. Gómez-Sanz, and J. Pavón. Model integration in agent-oriented development. *IJAOSE*, 1(1):2–27, 2007.

[10] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[11] J. J. Gómez-Sanz, R. Fuentes, and J. Pavón. Enabling rapid prototyping using decoupling of code skeletons and code generation process. *InfoComp, Journal of Computer Science*, 2007.

[12] A. Gravell and P. Henderson. Executing formal specifications need not be harmful. *Software Engineering Journal*, 11(2):104–110, Mar 1996.

[13] S. J. Mellor, B. M., and J. I. *Executable UML: A oundation for Modl-Drive Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] E. Mueller. Event Calculus Reasoning Through Satisfiability, 2004.

[15] A. Newell. The knowledge level (presidential address). *AI Magazine*, 2(2):1–20, 33, 1980.

[16] J. Pinto and R. Reiter. Reasoning about time in the situation calculus. *Annals of Mathematics and Artificial Intelligence*, 14(2):251–268, 1995.

[17] C. Ponsard, P. Massonet, A. Rifaut, J.-F. Molderez, A. van Lamsweerde, and H. T. Van. Early verification and validation of mission critical systems. *Electr. Notes Theor. Comput. Sci.*, 133:237–254, 2005.

[18] L. Steels. *The Artificial Life Route to Artificial Intelligence*, chapter Building Agents out of Autonomous Behavior Systems. Lawrence Arlbaum Associates, Inc., 1995.

[19] van Deursen A., P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[20] A. van Lamsweerde. Goal-oriented requirements engineering: a guided tour. *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262, 2001.

[21] L. S. Vygotsky. *Mind and Society*. Harvard University Press, 1978.

[22] S. Zimmerbaum and R. B. Scherl. Sensing actions, time, and concurrency in the situation calculus. In C. Castelfranchi and Y. Lespérance, editors, *ATAL*, volume 1986 of *LNCS*, pages 31–45. Springer, 2000.