Position Paper: Towards a Requirements-Driven Design of Ensemble-Based Component Systems

Ilias Gerostathopoulos, Tomas Bures and Petr Hnetynka
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25, 11800, Prague, Czech Republic
{iliasg,bures,hnetynka}@d3s.mff.cuni.cz

ABSTRACT

Although approaches that effectively address the distribution and dynamism of adaptive systems at a middleware level exist, the design of complex, ensemble-based systems still remains a significant challenge. This hinders the development of real-life applications based on the ensemble paradigm. A promising approach appears to be the coupling of proven low-level concepts with high-level ones, revisiting requirements modeling in the realm of ensemble-based systems. To this end, the goal of this paper is to point out the specific challenges related to the design of ensemble-based systems and show that classic requirements models and methods cannot be applied out-of-the-box in a requirements-driven design of ensemble-based applications. In response to this problem, a novel design method based on the iterative refinement of system requirements expressed by predicates on stakeholder's knowledge is discussed.

Keywords

Ensemble based, requirements engineering, system design

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

1. INTRODUCTION

Current trends in the area of information technology focus on the development of highly distributed systems comprised of sophisticated on-demand computing services, often characterized as *cloud systems*. A subset of such systems is the ones operating in the so-called *ad-hoc clouds*, i.e., in highly dynamic environments (typically over ad-hoc networks), where no guarantees regarding the availability and responsiveness of their constituting parts exist. Examples are systems of intelligent vehicle navigation, decentralized flight planning and healthcare monitoring. These systems feature a significant level of autonomy [24], which is connected with (self-) awareness [26] and (self-) adaptation [18] properties. At the same time, they feature a number of challenges in their development, as traditional software development approaches rely on static software architecture and static pre-defined behavior and fail to efficiently capture the dynamic architecture and support the overall development process.

In response to this problem, the new paradigm of *ensemble-based development* has been suggested to guide the development of large-scale adaptive systems operating in dynamic environments, termed Ensemble-Based Component Systems (EBCSs) [6]. These systems are typically composed of *autonomous service-components*, forming dynamic groups which encapsulate knowledge, interaction, and goals specific to the groups. These dynamic groups of components are termed *ensembles* [11].

The goal of this paper is to point out the specific challenges related to the design of EBCSs and suggest ways to deal with them. First, EBCSs are described (section 2), then the challenges are articulated (section 3) and finally two "classic" and one novel approach are assessed on how well they deal with the identified challenges (sections 4 and 5). The last two sections present key related work (section 6) and conclude (section 7).

2. ENSEMBLE-BASED COMPONENT SYSTEMS

Investigating ways to model and design systems based on the ensemble paradigm is the core of the European project ASCENS (Autonomic Service-Component ENSembles) [12]. A first attempt to formalize the concept of ensembles within ASCENS has led to the development of SCEL (Service-Component Ensemble Language) [9], a formal language for modeling component systems enabling them for further analysis and verification. Relying on the concepts of SCEL, DEECo (Dependable Emergent Ensembles of Components) component model [6, 14] has been conceived and is currently under development and refinement.

The goal of DEECo is to allow for building systems consisting of autonomous, self-aware, and adaptable components, which are implicitly organized in ensembles. To this end, DEECo suggests a slightly different way of perceiving a component than is common in component-based software engineering; i.e., as a self-aware unit of computation, relying solely on its local data that are subject to modification during the execution time. The whole communication process relies on data exchange among components (prescribed by ensembles), entirely externalized and automated within the

HotTopiCS'13, April 20–21, 2013, Prague, Czech Republic.

runtime. This way, the components have to be programmed as autonomous units, without relying on whether/how the distributed communication is performed, which makes them very robust and suitable for rapidly changing environments.

Apart from being perceived and articulated in terms of autonomous components and ensembles, EBCSs are built around three key ideas:

- The notion of *belief* and its explicit management have a central role. Every component in an EBCS operates upon its local "private" knowledge, which represents the component's view of the environment and of the other components. Since this knowledge is at any time subject to change by the runtime framework, which is responsible for mapping parts of one component's private knowledge body to knowledge bodies of other components, it is better viewed as the "belief" that a component preserves. In that sense, ensembles, being prescriptions of knowledge mappings between components, stand as the belief management mechanism.

- Component computation is performed *in isolation*. In EBCSs, there are no means for a component to explicitly communicate with others. Component communication is realized implicitly by knowledge exchange externalized from the components and performed by the underlying framework. Thus, every computation is necessarily performed within a component's boundaries, which strengthens the notion of component autonomy.

- Component bindings are *dynamic*. In EBCSs, there are no explicit bindings between components. Ensembles bind components implicitly by prescribing the appropriate knowledge exchange. However, ensembles are formed only when specific conditions hold in the system, not always. This dynamic nature of ensembles makes the architecture of EBCSs "emerge" during runtime.

As an example illustrating the main concepts of EBCSs, let us consider a system of intelligent vehicle navigation. The system consists of drivers, moving around a city in their "smart" vehicles. Drivers have to reach particular destinations within some time limits, which depend on their daily schedule. Vehicles are equipped with sensors of basic capabilities, e.g., monitoring the fuel and battery level of the car, but also more sophisticated ones, e.g., monitoring the traffic level along the route. Vehicles can only park and refuel in designated stations. They can also communicate with each

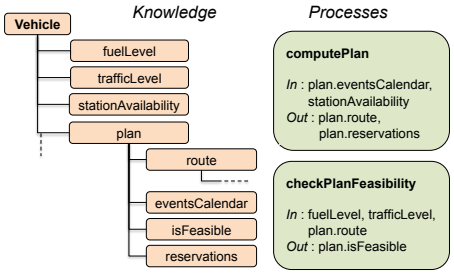

Figure 1: `Vehicle` component specification in DEECo.

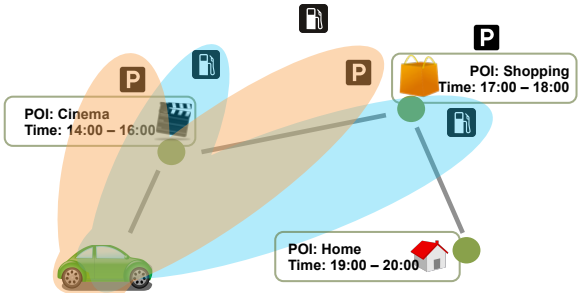

Figure 2: Possible ensembles between a `Vehicle` component and several `Parking/Refueling Station` components.

other and with the parking/refueling stations that lie within their transmission spectrum. No central coordination point is assumed; there is no global control or global planning. The whole system can be seen as a set of nodes which form dynamic communication networks (ad-hoc clouds) to serve a specific goal: vehicles should arrive at their destinations in time, leveraging the infrastructure resources in a close-to-optimal way.

When viewing the above system as an EBCS, the obvious candidates for (DEECo) components are vehicles, parking stations and refueling stations. For example, the `Vehicle` component can be specified in terms of its knowledge and processes as in Figure 1.

Possible (DEECo) ensembles are identified by looking at the different interaction scenarios among the components. For example, when a vehicle is close to a parking station, it may need to contact it to get informed about the available parking lots and reserve a lot if possible. This interaction is prescribed by one ensemble operating between a vehicle and (possibly multiple) parking stations, where the `Vehicle` obtains a belief over the lot availability information of the `Parking Stations`. A graphical representation of possible ensembles between a vehicle and parking or refueling stations is shown in Figure 2.

3. PROBLEM STATEMENT

It is clear that the number of different components and possible combinations of them in ensembles grows together with the magnitude of the system and the number and complexity of interactions we want to model. Consequently, even if we are able to *engineer* the above intelligent navigation system in terms of DEECo concepts, it remains challenging to *design* such a emergent system and retain some guarantees regarding its overall behavior and interactions.

This stems from the fact that DEECo concepts are rather low-level and focus primarily on supporting the implementation and deployment. They lack a broader system view that will take into account the system requirements and design alternatives. A broad, high-level view is crucial when dealing with systems of high complexity as it allows abstracting away from details of computation and interaction and reasoning about properties of the (distributed) system as a whole. Examples of interesting properties are performance-related ones, like communication overhead, information utilization, etc., and stability-related ones, like immunity to environmental changes, adaptability, robustness, etc.

Another issue is that it is problematic to map the architecture of the system (naturally comprising a number of components and ensembles) to the purpose it serves (its rationale). This is especially true for complex systems with numerous components and ensembles: in such cases tracing a low-level design decision, like the inclusion of a dynamic communication link in the system, back to its origin in the requirements analysis gets extremely difficult. At the same time, in such cases, specifying *why* an interaction has to take place is as important as specifying the interaction itself, as it allows for design justification and system predictability.

4. REQUIREMENTS MODELING

In order to be able to design and reason about an EBCS, we need to differentiate between stable and volatile information by obtaining a high-level view over the system. In this section, we will focus on approaches that capture the high-level behavior of a software system. For that, we need to draw our attention into the early phases of software development, such as the requirements analysis phase. It is thus necessary to examine effective approaches in requirements modeling and assess their applicability in the domain of EBCSs.

A useful abstraction in Requirements Engineering (RE) is proven to be the *system goal*. A goal can be defined as a prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some agents forming the system. Goal-Oriented Requirements Engineering (GORE) [19] is concerned with the identification of system requirements through the elicitation and analysis of system goals.

Another useful abstraction is that of the *agent*. Agents are active components, i.e., with a choice of behavior, which may restrict their behavior to ensure the constraints that they are assigned. In GORE, agents are assigned responsibility for achieving goals.

In the rest of the section, the two most prominent approaches in GORE, KAOS and Tropos/i* methods are presented and discussed.

4.1 KAOS

KAOS is a goal-oriented requirements engineering methodology with a rich set of formal analysis techniques. KAOS stands for Keep All Objects Satisfied [23]. It is grounded on the following main ideas:

- The notion of *goals* has a prominent role during the requirements acquisition and analysis processes, offering the common advantages of goal-oriented approaches in RE [19].

- *Formal methods* are used *when* and *where needed* for RE-specific tasks. This allows different levels of expression and reasoning: semi-formal for modeling and structuring goals, qualitative for selection among alternatives, and formal for more accurate reasoning. This is possible, as each element modeled in KAOS has, in general, a two-level structure: the *outer, semantic* layer where the concept is declared together with its attributes and relationships to other concepts and the *inner, formal* layer for formally defining the concept. Formal reasoning can be used e.g., for checking goal refinement [8] and goal operationalization [15], conflict management [21] and obstacle (hazard,threat) analysis [22].

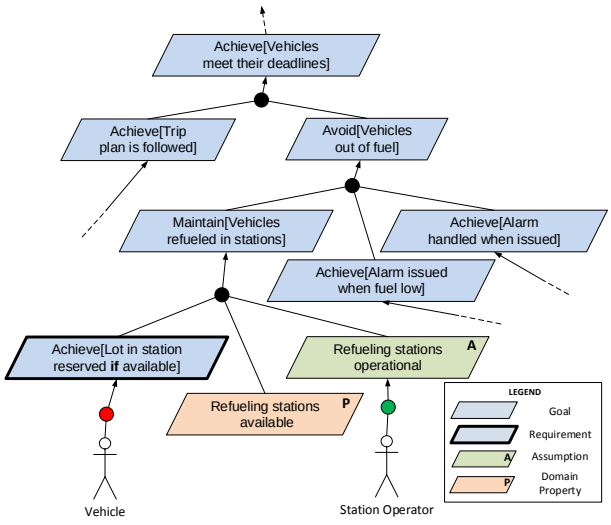

Figure 3: An excerpt of a KAOS goal model of the intelligent navigation case study.

A KAOS specification is a collection of complementary core models, which represent different views over the target system. In order to assess the applicability of KAOS models in the design of EBCSs, we will exemplify the process of deriving a KAOS specification on our intelligent navigation case study. The process spans four (practically interleaved) steps:

Goal elaboration step.

Goals are primarily obtained through the inspection of intentional keywords in natural language of stakeholders and by asking *why* and *how* questions about such statements. Goals are defined at different levels of abstraction: high-level goals capture global, strategic objectives; low-level goals capture local, technical objectives. After the elicitation of main goals, goals are organized into AND/OR refinement hierarchies with obvious semantics.

Goal refinement ends when every terminal goal is realizable by a single agent assigned to it. This means that the goal must be expressible in terms of conditions that are monitorable and controllable by the agent. In particular, a goal assigned to a software agent is a *software requirement*, whereas a goal assigned to an environment agent (e.g., a human agent) is an *expectation* or *assumption*.

Figure 3 depicts a possible goal refinement in the intelligent navigation case study, where the parent goal of having the vehicles refuel in the designated stations is decomposed into a requirement (vehicles reserve their places in the stations), an assumption (stations continue to operate) and a domain property (stations are available). Domain properties are descriptive statements (as opposed to prescriptive ones, like goals or assumptions) that are explicitly captured in the goal model and serve for checking its completeness.

As an example of a complete goal specification, Figure 4 depicts the goal that vehicles should reserve their places in the refueling stations. Apart from its semantic layer, the

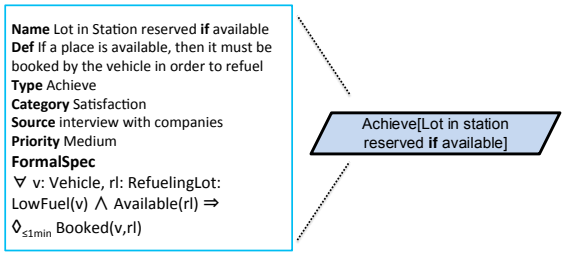

Figure 4: Formal specification of a goal in KAOS.

goal's formal layer is captured in real-time Linear Temporal Logic (LTL). This enables the automatic verification of the goal model.

Agent modeling step.

An *agent* is an active object of the system, which acts as processor for operations. Agents can be software entities, but also environment entities, like human agents, devices, etc. They appear in the system in order to handle some requirement or assumption assigned to them during goal elaboration. For example, in Figure 3, the `Vehicle` and `Station Operator` agents are introduced. There is no single correct agent assignment; as with choosing among alternative goal refinements, assigning terminal goals to agents represents a design choice.

Object modeling step.

Objects are things of interest in the system whose instances may evolve from state to state. An object is modeled as an *entity*, *association* or *event*, depending on whether it is independent, dependent or instantaneous, respectively. Objects are derived by traversing the goal model and inspecting which entities are concerned in every goal. In our case study, examples of objects are the `Refueling station` entity, the `Reserved` association and the `Alarm issued` event.

Operationalization step.

The functions the agents need to employ in order to *operationalize* (fulfill) their assigned requirements is defined in terms of *operations*. An operation is an input-output relation over objects (specifically, object instances), whereas the application of the operation defines object creation and object state transition. Operations are derived both by goal fluents and from interaction scenarios, identified during requirements acquisition. Operations are declared by signatures over objects and have pre-, post-, and trigger conditions, specified in real-time LTL. As an example from our case study, a `Reserve lot` operation receiving a `Reservation request` object and creating the `Lot reservation` object could operationalize the "Lot in station reserved if available" requirement (Figure 3).

4.1.1 Discussion

Overall, KAOS is a well-developed methodology which exhibits the virtues of goal-oriented analysis in the field of requirements engineering. It successfully separates the stable (goals, assumptions, properties) from the volatile information (agents, operations) and provides a high-level view over the target system.

Another strong point of KAOS methodology is its formal reasoning support. When goals, domain properties, assumptions and operations are specified formally, the underlying models can be checked for consistency and completeness, typically by employing a SAT solver or theorem prover [16]. A semi-formal approach is also provided: during goal refinement and goal operationalization, the analyst can rely on instantiations of *formal refinement patterns* (extracted from a patterns catalogue) that are proven once and for all [8, 15].

+	1. captures the (intended) system behavior at a high level 2. allows for automatic formal reasoning
−	1. does not align requirements with architecture 2. is intended for requirements analysis and documentation, not system design

Table 1: KAOS positive and negative points.

A limitation of KAOS is that it does not provide a connection between requirements and architecture. Preliminary efforts towards this direction can be found in [20], where a process of deriving an architectural draft from goal, agent and operation models, is proposed. This draft is then iteratively refined based on instantiation of pre-defined architectural styles and patterns. This is a preliminary and rather generic attempt towards bridging the well-known gap between requirements and architecture [20]. When designing EBCSs, a tailored approach, dealing with the specific domain intricacies (high distribution, dynamism), seems more viable.

The main problem in directly applying KAOS towards meeting our goal of designing EBCSs is that the (classic) outcome of KAOS analysis is a Software Requirements Specification (SRS) document. While this serves the needs of requirements analysis, a successful design of complex, ensemble-based systems demands a mapping of the high-level models to lower-level ones and eventually to implementation artifacts.

The applicability of KAOS in the design of EBCSs is summarized in Table 1.

4.2 Tropos and i*

Tropos [4] is a methodology for building agent-oriented software systems that uses the i* modeling notation [25] (i-star refers to *distributed intentionality*). Tropos is based on the following key ideas:

- The notion of *agent* and related notions (e.g., goals, plans) are used in all software development phases, from early requirements down to actual implementation. To qualify as an agent, a software (or hardware) system is often required to have properties such as autonomy, social ability, reactivity, proactivity and rationality.
- The *early phases* of requirements analysis, i.e., the phases which precede the prescriptive requirements specification of the system-to-be, are considered equally important to the later phases and are thus elaborated in detail. This allows for a deeper understanding of the environment (organizational context) where the software must operate and facilitates the early resolution of conflicts between stakeholders.

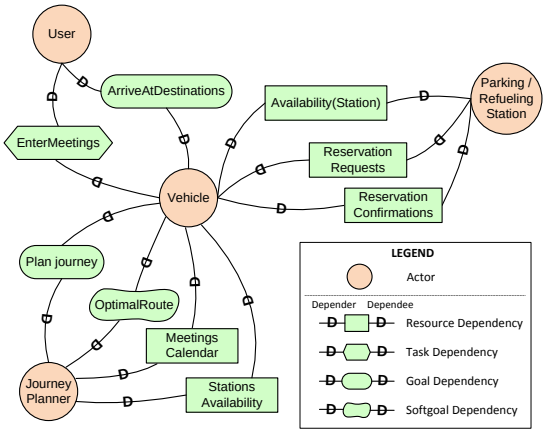

Figure 5: A possible i* Strategic Dependency model of the intelligent vehicle navigation case study.

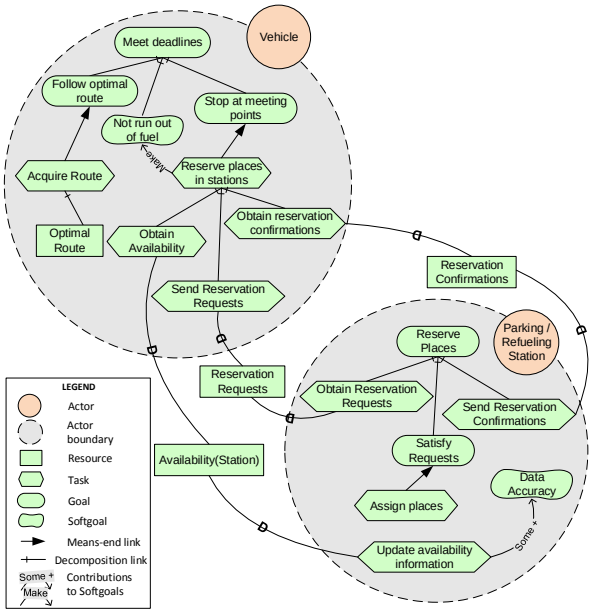

Figure 6: A possible i* Strategic Rationale model for `Vehicle` and `Parking/Refueling Station` actors.

The Tropos methodology spans four levels:

Early requirements.

The early requirements phase is concerned with the understanding of a problem by studying its organizational setting (the *system-as-is*). The output of this phase is an i* *Strategic Dependency* (SD) and an i* *Strategic Rationale* (SR) model. The SD model captures the relevant actors and their interdependencies in terms of goals to be achieved, tasks to be performed and resources to be furnished. The SR model determines through a means-end analysis how the goals can be fulfilled through contributions of other actors.

Late requirements.

During late requirements phase the *system-to-be* (target system) is described within its operational environment. The output of this phase is a requirements specification in the form of SD and SR models (refined versions of the early requirements phase models), which describes all functional and non-functional requirements of the system-to-be. The difference from the early requirements phase is that now the software to be developed comes into the picture as one or more intentional actors.

Figure 5 depicts a possible SD graph of our intelligent navigation case study at the late requirements phase. As depicted, the `User` actor depends on `Vehicle` to arrive at his destinations in time and, vice-versa, `Vehicle` depends on the fact that `User` will follow the routine of entering the meetings in the calendar. Dependencies exist between the `Vehicle` and the `Parking/Refueling Station` (modeled together for brevity) as well. `Journey Planner` has been introduced at this phase as part of the software to be developed. `Journey Planner` is delegated the responsibility of computing an (optimal) journey plan.

A means-end analysis of the `Vehicle` and the `Parking/Refueling Station` actors is depicted in the SR model of Figure 6. Specifically, the `Vehicle`'s goal of meeting the scheduled deadlines is decomposed into sub-goals, which are operationalized into tasks. As an example, the goal of being able to stop at the meeting points is satisfied by performing (in advance) reservations at the parking/refueling stations, which in turn is decomposed into the tasks of obtaining the station availability information, requesting a place in the relevant stations, and receiving reservation confirmations. At the `Station`'s side, the identified goals and tasks are decomposed in a similar manner.

Architectural design.

Architectural design defines the system's global architecture in terms of sub-systems, interconnected through data and control flows. The SD and SR models derived from previous levels are further refined. The refinement – inclusion and removal of actors and dependencies – is determined by a quality analysis (analysis and refinement of softgoals). The quality analysis guides the selection between alternative architectural styles from a catalogue of organizational styles (pre-defined in terms of the i* concepts of actors, dependencies, goals and tasks) [10]. For example, if during the requirements phase of our intelligent navigation case study the "response time" softgoal was identified, we could choose an organizational style which ensures low response time, e.g., the "joint venture" or the "pyramid" style.

After the refinement of the actor and goal models, the *capabilities* needed by the actors to fulfill their goals and plans are identified by inspecting the extended SD (actor) diagram, presuming that each actor's dependency relationship can give place to one or more capabilities triggered by external events.

As a final step, a set of *system agents* is identified and for each of them one or more capabilities are assigned. In general, the process of assigning capabilities to agents is not unique and depends heavily on the designer's view of the system (in terms of agents). Nonetheless, Tropos offers a set of pre-defined social patterns recurrent in multi-agent literature like `Bidding`, `Broker`, `Matchmaker`, `Embassy`, etc., to guide this process [10].

Detailed design.

Detailed design deals with the specification of the agents' micro-level, that is, the agents' behavior and communication. During detailed design, multiple *capability* and *plan diagrams* are created. Agent-UML (AUML) sequence diagrams are also employed to specify the interaction protocols.

Following the detailed design, a *concrete implementation* of the system is produced by using one of the agent-oriented development environments. For example, JACK Intelligent Agents [2] can be employed, which stands as a reification of the conceptual Belief-Desire-Intention (BDI) [17] agent model in Java programming language.

4.2.1 Discussion

The main contribution of Tropos methodology is that it tries to align requirements analysis with system design and implementation. Its novel idea is to base such an alignment on early requirements concepts, such as actors, goals and plans, rather than implementation-level concepts, like classes and methods. A strong point of Tropos is that it provides a small and manageable set of knowledge level notions, which are used throughout the software development process.

At the same time, the biggest shortcoming of this methodology is the plethora of design phases and models (actor/goal models, capability/plan/sequence models, BDI agent model), which complicates the design process. Moreover, the transition between the phases is in most cases manual and relies heavily on subjective design decisions. In the design of complex EBCSs the automatic transition between abstraction levels is a necessity and cannot be overlooked.

To conclude, the direct application of Tropos method in the design of EBCSs would eventually result into mapping one or more system agents to (DEECo) components and agent communication to ensembles. However, current agent development frameworks [2, 3] assume static bindings between the system actors and therefore fail to cope with the dynamic, emergent architecture (captured in terms of ensembles), which is one of the key characteristics of EBCSs.

The applicability of Tropos in the design of EBCSs is summarized in Table 2.

+	1.	aligns the requirements phase with architecture and implementation phases
	2.	preserves a manageable set of concepts throughout the software development phases
−	1.	comprises a number of models with manual mappings between them
	2.	does not cope with the emergent architecture of EBCSs

Table 2: Tropos positive and negative points.

5. TOWARDS AN ENSEMBLE-BASED DESIGN METHOD

Having evaluated KAOS and Tropos methods, it is clear that they are not directly applicable in the design of EBCSs. On their background, in this section we will describe a novel design method termed Predicate Refinement Method (PRM) [5], which employs similar concepts. We will also justify why PRM successfully deals with the intricacies of EBCSs, by accounting for what is missing in KAOS and Tropos.

PRM builds upon the idea of iterative refinement of system specification, employed in goal-oriented requirements engineering. The main goal of PRM is to complement DEECo low-level concepts with design-level abstractions that will allow for a) design-time reasoning, and b) automatic preparation of DEECo artifacts.

PRM is based on capturing the high-level system goals expressed in terms of propositional claims (predicates). It consists of three levels (phases): *system level* design, *ensemble level* design, and *component level* design, followed directly by implementation.

As a starting point, at the system level, PRM elaborates on the questions "which (global) goals must be achieved?" and "which system attributes must be maintained?". The next question is "who is responsible for achieving/maintaining these attributes?". Answering these questions yields the system's initial *stakeholders* and interaction *predicates*.

A stakeholder is a participant/actor of the system. Each stakeholder is defined in particular by its knowledge (i.e., its domain-specific data). For example, when applying the method in the running case study (Figure 7), the `Vehicle` and `Parking/Refueling Stations` stakeholders are identified. At this level, the system goals are expressed by predicates over the stakeholders and their knowledge; each involvement of a stakeholder in an predicate is a *role* of the stakeholder in the predicate. Unlike i*, PRM does not capture the dependencies of actors at a strategic level, but at the lower level of data that has to be exchanged and "believed"; stakeholders contribute their knowledge (interfaced by roles) for the assessment of each predicate they take a role in. Predicates represent system properties that should hold over the whole system lifetime. In an idealized system, predicates would become system *invariants*, whereas in a real system predicates should hold "frequently enough". As opposed to goals in KAOS, predicates have a more descriptive nature, and typically refer to the present state, not some future actions.

After identifying all top-level predicates, the process continues into refining them into sets of sub-predicates. The refinement is essentially an AND-decomposition, with the conjunction of sub-predicates implying the parent predicate. The iterative refinement process in PRM ends once all the leaf predicates are directly mappable to DEECo computational or communication semantics, that is, to component and ensemble processes. In particular, a predicate needs no further decomposition when a) it involves a single stakeholder and can be ensured by manipulation of this stakeholder's knowledge (via an underlying component process) – *local predicate* – or b) the predicate involves exactly two stakeholders and can be ensured by mapping one stakeholder's knowledge part(s) to the other (via an underlying knowledge exchange mechanism) – *exchange predicate*.

Figure 7 depicts a partial decomposition tree of the top-level predicate of having all destinations visited by the `Vehicle`. This predicate is refined into three "necessities", independent to each other: the necessity a) to have a feasible plan, b) to follow it, and c) to have places in stations reserved accordingly. Following the plan's route is essentially a local predicate, as it involves the `Vehicle` stakeholder only. In contrast, the predicate of having a correct belief over the reservation requests (Figure 7) is an example of an exchange

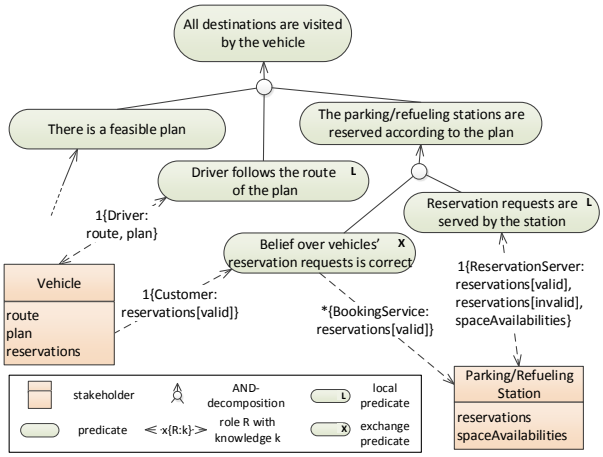

Figure 7: A partial decomposition of the main system predicate in the intelligent mobility case study.

predicate, as it involves both the `Vehicle` and the `Parking/Refueling Station` stakeholder.

At the next phases of PRM, the predicates are transformed to precise system specifications. Specifically, during *ensemble level design*, the exchange predicates are turned into DEECo ensembles, by specifying the necessary condition for knowledge exchange along with the information of which knowledge parts are going to be exchanged. In *component level design*, finally, stakeholders are turned into DEECo components, comprising the stakeholders' knowledge and "operationalizing" (in terms of KAOS) the local predicates they are involved in by means of (DEECo) processes.

Compared to PRM, KAOS and Tropos address well the high-level modeling, however naturally they do not provide constructs for alignment with the emergent architecture of EBCSs. This is summarized in Table 3. Further, to complete this picture, we evaluate below in more detail how PRM addresses the three fundamental characteristics of EBCSs.

Belief handling.

The semantics of the *belief* that components in EBCSs preserve about the other components and the environment is what remains stable throughout the system phases (over the whole lifetime of the system) and provides an effective way to reason about global properties of interest. In PRM, we explicitly capture this semantics at a higher level by modeling the knowledge flow in terms of knowledge that has to be distributed and "believed". Moreover, the mapping of exchange predicates to ensembles provides a convenient belief managing mechanism. Although the concept of belief is not new (agent-based approaches like Tropos employ the same concept in the design of agent-based architectures, like BDI), the novelty of PRM lies in featuring belief as the base abstraction that aligns requirements with architecture and implementation phases.

Isolated computation.

In order for component autonomy to be reified, components in DEECo perform their tasks in isolation. This is reflected in the design by introducing local predicates, which are mapped to component processes as the process progresses. In that sense, PRM captures both the requirements of the whole system (higher-level predicates) and of isolated components (local predicates), in contrast with requirements elicitation a là KAOS, which is necessarily system-wise.

	KAOS	Tropos	PRM
high-level modeling	+	+	+
req/ments & architecture alignment	–	+	+
dynamic, emergent architecture support	–	–	+

Table 3: A comparison of PRM with KAOS and Tropos methods.

Dynamic component links.

Dynamic component links, reified by ensembles in DEECo, provide the way to deal with the emergent architecture of EBCSs. In PRM, dynamic links are reflected by exchange predicates. This property of EBCSs is not directly supported either by KAOS or Tropos methods.

6. RELATED WORK

As we are not aware of any work that combines the ensemble paradigm with goal-oriented requirements analysis to devise a design method for EBCSs, in this section we will refer to general approaches towards the requirements modeling and design of EBCSs. We have already extensively described, in section 4, two prominent approaches in goal-oriented modeling and design, namely KAOS and Tropos/i* methods.

Recent work in requirements modeling and in particular targeting the domain of EBCSs has been carried out within the scope of the ASCENS project and has been integrated into Statement of the Affairs (SOTA) [1], General Ensembles Model (GEM)[13] and POEM [12] models.

SOTA is concerned with the overall domain and the requirements of the system. The key idea is to abstract the behavior of a system with a single trajectory through a *state space* (or state-of-the-affairs space), which represents the set of all possible states of the system at a single point of time. Similarly to PRM, the requirements of a system in SOTA are captured in terms of goals. A goal is an area of the SOTA space that a system should eventually reach, and it can be characterized by a goal pre-condition, its post-condition, and utilities (non-functional requirements or constraints). To support adaptation, SOTA relies on the goal model to help understand according to which (self-adaptive) architectural scheme a system should be architected so goals can be achieved [7].

For a more detailed specification of behaviors and goals, POEM [12] model has been proposed and is currently at a rather preliminary stage. Finally, GEM [13] stands as a formal system model which represents ensembles as relations that describe the complete behavior of the ensemble over its lifetime. GEM is intended to serve as a semantic foundation for various kinds of calculi and formal methods that often have a particular associated logic.

7. CONCLUSION

In this paper, the characteristics of systems based on the ensemble paradigm have been described. The challenges related to the design of EBCSs have also been identified. A successful design method should be able to capture the high-level objectives of the system under consideration in a model amenable to design-time analysis, and, at the same time, simplify the transition towards architecture-level models, like DEECo model. As possible ways to construct models of the system at a high-level we have looked into two prominent methods in requirements modeling, KAOS and Tropos/i*. They both possess their strong points but also feature serious limitations, when employed in the design of EBCSs. In response to that, a novel method based on the iterative refinement of system requirements expressed by predicates on stakeholders' knowledge was outlined. The novelty of the proposed method lies in reasoning along the line of what needs to hold in the system at every time instant (predicates), instead of what needs to be performed (actions) or achieved (goals).

As future work, we plan to work on a prototype implementation of PRM, which will exhibit the advantage of deriving the system components and ensembles from requirements in a (semi-) automatic way. We also plan to formalize the concepts of predicate and predicate refinement in such way that will allow for formal verification of the design model.

8. ACKNOWLEDGMENTS

This work is a part of RELATE project supported by the European Commission under the Seventh Framework Programme FP7 with Grant agreement no.: 264840ITN.

9. REFERENCES

[1] D. B. Abeywickrama, N. Bicocchi, and F. Zambonelli. SOTA: Towards a General Model for Self-Adaptive Systems. In *Proc. of WETICE '12*, pages 48–53. IEEE CS, 2012.

[2] Agent Oriented Software. JACK Intelligent Agents Manual, R. 5.3. http://www.agent-software.com, 2005.

[3] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

[4] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[5] T. Bures et al. Language Extensions for Implementation-level Conformance Checking. ASCENS Deliverable D1.5, 2012. Available: http://www.ascens-ist.eu/deliverables.

[6] T. Bures et al. DEECo – an Ensemble-Based Component System. Tech. Rep. D3S-TR-2013-02, D3S, Charles University in Prague, 2013. Available: http://d3s.mff.cuni.cz/publications.

[7] G. Cabri, M. Puviani, and F. Zambonelli. Towards a Taxonomy of Adaptive Agent-based Collaboration Patterns for Autonomic Service Ensembles. In *Proc. of CTS '11*, pages 508 –515. IEEE CS, 2011.

[8] R. Darimont and A. van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. *SIGSOFT Softw. Eng. Notes*, 21(6):179–190, Oct. 1996.

[9] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-based Approach to Autonomic Computing. In *Proc. of FMCO '11*, volume 7542 of *LNCS*, pages 25–48. Springer, 2012.

[10] P. Giorgini et al. A Goal-Based Organizational Perspective on Multi-Agents Architectures. In *Proc. of ATAL '01*, pages 128–140. ACM, 2001.

[11] M. Holzl et al. Engineering of Software-Intensive Systems: State of the Art and Research Challenges. In *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 1–44. 2008.

[12] M. Holzl et al. Engineering Ensembles: A White Paper of the ASCENS Project. ASCENS Deliverable JD1.1, 2011. Available: http://www.ascens-ist.eu/.

[13] M. Holzl and M. Wirsing. Towards a System Model for Ensembles. In *Festschrift in honor of Carolyn Talcott*, volume 7000 of *LNCS*. Springer, 2011.

[14] J. Keznikl, T. Bures, F. Plasil, and M. Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *Proc. of WICSA/ECSA '12*. IEEE CS, 2012.

[15] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. *SIGSOFT Softw. Eng. Notes*, 27(6):119–128, 2002.

[16] C. Ponsard, P. Massonet, J. F. Molderez, A. Rifaut, A. van Lamsweerde, and H. T. Van. Early Verification and Validation of Mission Critical Systems. *Form. Methods Syst. Des.*, 30(3):233–247, June 2007.

[17] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *Proc. of ICMAS '95*, pages 312–319. The MIT Press, 1995.

[18] M. Salehie and L. Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[19] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proc. of RE '01*. IEEE CS, 2001.

[20] A. van Lamsweerde. From System Goals to Software Architecture. In *Proc. of FSM '03*, volume 2804 of *LNCS*, pages 25–43. Springer, 2003.

[21] A. van Lamsweerde, R. Darimont, and E. Letier. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Trans. Softw. Eng.*, 24(11):908 –926, Nov. 1998.

[22] A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, Oct. 2000.

[23] A. van Lamsweerde and E. Letier. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In *Proc. of RISSEF '12*, volume 2941 of *LNCS*, pages 325–340. Springer, 2004.

[24] E. Vassev and M. Hinchey. The Challenge of Developing Autonomic Systems. *Computer*, 43(12):93 –96, Dec. 2010.

[25] E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *Proc. of RE '97*, pages 226–. IEEE CS, Jan. 1997.

[26] F. Zambonelli et al. On Self-Adaptation, Self-Expression, and Self-Awareness in Autonomic Service Component Ensembles. In *Proc. of SASOW '11*, pages 108 –113. IEEE CS, 2011.