Model Checking Goal-Oriented Requirements for Self-Adaptive Systems

Dhaminda B. Abeywickrama
Department of Science and Engineering Methods
University of Modena and Reggio Emilia
Via G. Amendola 2, 42122, Reggio Emilia, Italy
dhaminda.abeywickrama@gmail.com

Franco Zambonelli
Department of Science and Engineering Methods
University of Modena and Reggio Emilia
Via G. Amendola 2, 42122, Reggio Emilia, Italy
franco.zambonelli@unimore.it

***Abstract*—To deal with the increasing complexity and uncertainty of software systems, novel software engineering models and tools are required to make such systems self-adaptive. As part of ongoing research, we investigate various models, schemes and mechanisms to model and engineer self-adaptation in complex software systems. To this end, we have defined SOTA (State of the Affairs) as a general goal-oriented modeling framework for the analysis and design of self-adaptive systems. In this paper, by transforming the conceptual SOTA model into an operational one, we show how SOTA can be an effective tool to perform an early, goal-level, model checking analysis for adaptive systems. This allows the developers of complex self-adaptive systems to validate the actual correctness of the self-adaptive requirements at an early stage in the software life-cycle. The approach is explored and validated using a case study in the area of e-mobility.**

***Keywords*-self-adaptive systems; model checking; goal-oriented requirements engineering; software architecture**

I. Introduction

Software systems are becoming increasingly complex, decentralized, and called to operate in open-ended and dynamic environments. As a consequence, it is becoming difficult and economically unbearable to develop, deploy and manage them with the necessary degree of reliability and availability. New software engineering models and tools are required to make software systems autonomic [1], i.e. capable of autonomously self-adapt their behavior and structure to accommodate changes in their operational environment without suffering malfunctionings [2].

In the context of the ASCENS European Project (www.ascens-ist.eu), we are investigating various models, schemes and mechanisms that can be used to express self-adaptation at various levels [3]. The ultimate goal is to provide a uniform set of conceptual and practical tools to guide developers at the level of abstract modeling, verification and implementation. As a first step towards this objective, we have defined the *SOTA* (State Of The Affairs) conceptual model, a general goal-oriented modeling framework for analyzing the self-adaptation requirements of adaptive systems.

Goal-oriented approaches to requirements engineering are increasingly adopted in the area of autonomic and self-adaptive systems [4], [5], [6]. In fact, the notion of goals – intended as the state of the affairs that a system has to achieve – effectively captures the observably intentional properties of self-adaptive systems. By bringing together the current lessons of goal-oriented requirements engineering and multi-dimensional context-modeling [7], SOTA proposes to serve as a comprehensive conceptual tool to guide across all phases of self-adaptive systems engineering.

The specific contribution of this paper is to show how SOTA can serve as a framework for eliciting operational descriptions of the self-adaptive system, upon which formal techniques to perform an early, goal-level, model checking analysis – i.e. the application of automated techniques for verifying the formal correctness of elicited requirements [8] – can be effectively applied. Although there are several approaches on the goal-oriented requirements engineering of self-adaptive systems (e.g. [9], [6], [5], [10], [4]), we emphasize that these works provide little support for automated verification analysis techniques such as model checking.

Software requirements in SOTA can be effectively distinguished into functional ones (called goals, and representing state of the affairs that a system or its components have to eventually achieve) and non-functional ones (called utilities, and representing state of the affairs that have to be preserved while achieving). Starting form their identification, operational representations of both goals and utilities can be derived systematically, in terms of tasks for goals and utilities. Then, such operationalized goals and utilities are assigned to specific event-based service component behavioral models, upon which a model checking approach (based on an operational, event-based, untimed and asynchronous model of labeled transition systems) can be applied.

As we will discuss with the help of a case study in the area of e-mobility [11], our approach supports the development of self-adaptive systems by allowing to validate their actual correctness before implementation later in the software life-cycle. The specific advantages of the proposed approach are related to exploiting the goal-oriented modeling of SOTA where it provides a systematic method for modeling real-world goals of a system, such as a refinement hierarchy, conflicts and exceptions handling; thus gradually deriving specifications that satisfy the goals. Also, the approach can exploit event-based systems for automating the for-

mal analysis of software architecture specifications, and for supporting software architecture design and program verification and testing. More specifically model checking is applied to: identify any *incompleteness* of the SOTA goal-oriented requirements model by checking for single goal or utility operationalization and higher-level goal or utility satisfaction; identify *inconsistencies* and *implicit requirements* that can be detected as deadlocks; and *animate* the goals-oriented models using the standard animation and simulation features of the LTSA.

The counterexample traces generated and the deadlocks detected in the model checking process are used to iteratively refine and improve the SOTA requirements model, building a specification that is correct.

Paper organization: Section II presents our SOTA conceptual model. The e-mobility case study with a running example is described in Section III. In Section IV, we discuss our model checking approach and Section V examines the actual verification using the running example. A discussion of our approach and evaluation are provided in Section VI. Section VII highlights key related work and Section VIII concludes this paper.

II. The SOTA Conceptual Model

The SOTA conceptual model builds on the most assessed approaches to goal-oriented requirements engineering [6], [12]. For modeling the adaptation dimension, SOTA integrates and extends recent approaches on multidimensional modeling of context such as the Hyperspace Analogue to Context (HAC) approach [7]. In particular, such generalization and extensions try to account for the general needs of dynamic self-adaptive systems and components.

The *state of the affairs* represents the state of everything in the world in which the system lives and executes that may affect its behavior and that is relevant with regards to its capabilities of achieving. We could also say that such state of affairs is the *context* of the system. Context has been defined as "any information that can characterize the situation of an entity" [13].

The current *State Of The Affairs* $S(t)$ at time t, of a specific entity e (let it be an individual component or an ensemble) can be described as a tuple of n s_i values, each representing a specific aspect of the current situation:

$$S(t) =< s_1, s_2, \ldots, s_n >$$

As the entity executes, S changes either due to the specific actions of e or because of the dynamics of e's environment. Thus, we can generally see this evolution of S as a movement in a virtual n-dimensional space $\mathbf{S}$ (see Fig.1):

$$\mathbf{S} =< \mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n >$$

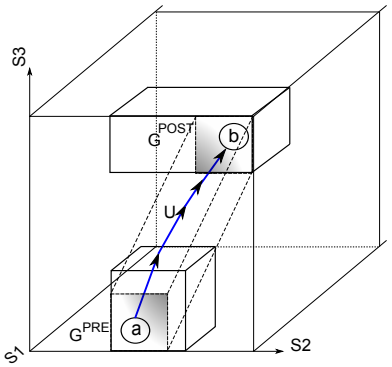

Figure 1: The trajectory of an entity in the SOTA space, starting from a goal precondition and trying to reach the postcondition while moving in the area specified by the utility

Or, accordingly to the terminology of dynamical systems modeling, we could consider $\mathbf{S}$ as the phase space of e and its evolution/execution as a movement in such phase space. Such movements can be endogenous, i.e. induced by actions within the system itself, or exogeneous, i.e. induced by external sources. The existence of exogeneous transitions is particularly important to account for, in that the identification of such movements (better of which dimensions of the SOTA space can induce them) enables identifying what can be the non-internally controllable external factors requiring adaptation.

A *goal* by definition is the eventual achievement of a given state of the affairs. Therefore, in very general terms, a specific goal G_i for the entity e can be represented as a specific point, or more generally of a specific area, in such space. That is:

$$G_i =< A_1, A_2, \ldots, A_n >, A_i \subseteq \mathbf{S}_i$$

Getting more specific, a goal G_i got an entity e may not necessarily be always active. Rather, it can be the case that an entity has a goal activated only when specific conditions occur. In these cases, it is useful to characterize a goal in terms of a precondition G_i^{pre} and a postcondition G_i^{pre}, to express when the goal has to activate and what the achievement of the goal implies. Both G_i^{pre} and G_i^{post} represent two areas (or points) in the space $\mathbf{S}$. In simple terms: when an entity e finds itself in G_i^{pre} the goal gets activated and the entity should try to move in $\mathbf{S}$ so as to reach G_i^{post}, where the goal is to be considered achieved (see Fig.1). Clearly, a goal with no precondition is like a goal whose precondition coincides with the whole space, and it is intended as a goal that is always active.

As goals represent the eventual state of the affairs that a system or component has to achieve, they can be considered functional requirements. However, in many cases, a system should try to reach its goals by adhering to specific con-

straints on how such a goal can be reached. By referring again to the geometric interpretation of the execution of an entity as movements in the space $\mathbf{S}$, one can say that sometimes an entity should try (or be constrained) to reach a goal by having its trajectory be confined within a specific area (see Fig.1). We call these sorts of constraints on the execution path that a system/entity should try to respect as *utilities*. This is to reflect a nature that is similar to that of non-functional requirements.

As goals a utility U_i can be typically expressed as a subspace in $\mathbf{S}$, and can be either a general one for a system/entity (the system/entity must always respect the utility during its execution) or one specifically associated to a specific goal G_i (the system/entity should respect the utility while trying to achieve the goal). For this latter case, the complete definition of a goal is thus:

$$G_i = \{G_i^{pre}, G_i^{post}, U_i\}$$

In some cases, it may also be helpful to express utilities as relations over the derivative of a dimension, to express not the area the trajectory should stay in but rather the *direction* to follow in the trajectory (e.g. try to minimize execution time, where execution time is one of the relevant dimension of the state of affairs). It is also worth mentioning that utilities can derive from specific system requirements (e.g. for a vehicle trying to reach a destination while minimizing fuel consumption) or can derive from externally imposed constraint (e.g. trying to reach a destination in respect of existing speed limitations).

A complete definition of the requirements of a system-to-be thus implies identifying the dimensions of the SOTA space, defining the set of goals (with precondition and postcondition, and possibly associated goal-specific utilities) and the global utilities for such systems, that is, the sets:

$$\mathbf{S} =< \mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n >$$
$$\mathbf{G} = \{G_1, G_2, \ldots, G_m\}$$
$$\mathbf{U} = \{U_1, U_2, \ldots, U_p\}$$

Of course, during the identification of goals and utilities, it is possible to associate goals and utilities locally to specific components of the system (e.g. for the e-mobility case study, there are goals and utilities associated to vehicles) and globally to the system as a whole (e.g. in e-mobility, the minimization of pollution is a global utility of the system rather than one associated to local vehicles). Thus, the above sets can be possibly further refined by partitioning them among local and global ones.

The SOTA conceptual model can be useful for three main reasons.

- First, a sound requirements engineering activity is necessary to assess and improve requirements identification. This is by deriving an operational model for SOTA that allows to perform model checking of goals and utilities.
- Second, SOTA modeling is useful to derive the knowledge and awareness requirements of the system.
- Third, SOTA helps to understand according to which architectural scheme a system should be architected so goals can be adaptively achieved.

Out of these, this paper focuses and addresses on the first use while the assessment of the latter two is still in progress.

III. The E-Mobility Case Study

The e-mobility case study (individual planning vehicles scenario [11]) of the ASCENS project is used to explore and validate our approach. A subset of this case study is provided to be used as a running example in the following sections.

The main challenges faced by the next generation e-vehicles are urbanization, resource restrictions, and flexible and diverse user demands [11]. In this context, the ASCENS project proposes a user-centric, constraint-based, automated travel planning system for electronic vehicles. To this end, it integrates appointment scheduling, mobility planning and resource scheduling. By exploiting knowledge about the current and future states of the vehicle-user-infrastructure network, the system schedules a daily travel plan for a user. The main resource restrictions are parking space, energy and time. This mobility system is conceptually modeled as an ensemble of entities such as a user, a vehicle and a parking lot operator. Each of these can be modeled in terms of entities having goals, and there are utilities (at individual or global level) related to how such goals can be achieved.

There are several vehicle service components competing for parking spaces and charging lots at a car park. The specific SOTA entities or components that are of interest are vehicles, users (drivers) and the parking lot operator. The operational model created in this study are based on finite state machines or simply processes, which represent the SOTA entities or components. There are multiple processes in the execution of a concurrent event-based model, and the transitions in the processes can be of exogeneous or endogenous type. Against this background, we can identify the following service component processes in the model: the arrivals process (`VEHICLEARRIVALS`), the departures process (`VEHICLEDEPARTURES`), the process that controls access to parking space (`PARKINGSPACECONTROL`), and the process that controls access to charging lots (`CHARGINGLOTSCONTROL`) (Fig. 2).

Next we can identify the goals and utilities that are of interest in the model. To this end, a goal called `ASSIGNPARKINGSPACE` can be considered for the vehicle component, and a goal-specific utility called `CHARGINGLOTSAVAILABLE` for the same component. Here we are only associating goals and utilities locally

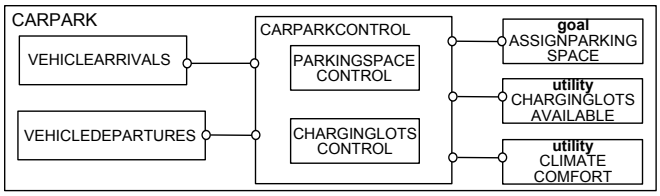

Figure 2: Car park model with a goal and two utilities

to specific components of the system and not globally to the system as a whole. The ASSIGNPARKINGSPACE goal is concerned with assigning parking space at the car park. The CHARGINGLOTSAVAILABLE utility checks whether there are charging lots available at the car park. As stated in Section II, utilities can constrain how a goal is achieved, that is by confining its execution path or trajectory. CHARGINGLOTSAVAILABLE is an example of a utility which expresses the area the trajectory should not stay in the space (i.e. not consider a car park where charging lots are not available). The vehicle component needs to respect this utility while trying to achieve its goal.

We can characterize the ASSIGNPARKINGSPACE goal in terms of a precondition and a postcondition to express when the goal has to activate and what the achievement of the goal implies. For example the precondition can be the utility for checking the availability of charging lots and the postcondition is the actual assigning of the parking space.

The ASSIGNPARKINGSPACE goal and the CLIMATECOMFORT utility can form a higher-level goal for a user component, which is concerned with achieving user preferences (see the User Preferences goal in Fig. 4). In SOTA terms, this is defining a set of goals (with pre and postconditions and goal-specific utilities). The CLIMATECOMFORT utility is an example which specifies the direction to follow in the trajectory in the space (i.e. climate comfort needs to be maintained or preserved as eco or maximal until the trip ends).

The scenarios discussed are further exemplified for model checking purposes later in the paper.

Next by turning the SOTA conceptual model into an operational one, we show how SOTA can be an effective tool to perform an early, goal-level, model checking analysis for adaptive systems, which is the focus of the current paper.

IV. The Model Checking Approach

In this section, our model checking process is discussed.

Our approach is based on formal verification, validation and simulation techniques provided by the model checker Labeled Transition System Analyzer (LTSA) [14] and its process calculus Finite State Processes (FSP). The formalism that we use to model goals and utilities is Fluent Linear Temporal Logic (FLTL) of the LTSA. Fluents have been used to provide a uniform framework for specifying properties that combine event and state-based predicates, and to automatically verify their satisfaction by an event-based model [15].

Our overall model checking process is divided into four main stages (Fig. 3).

A. Requirements Modeling with i*

First, early requirements of the organization context are described using the concepts provided in the i* framework, such as actors and intentional elements [16]. Actors are active entities of the organizational context who perform actions to achieve their goals, for example vehicle drivers and the parking lot operator in e-mobility. We use a strategic dependency model to describe the dependencies among various actors (i.e. relationships between two actors called a depender and a dependee). Intentional elements (e.g. *goals*, *soft goals*, *tasks* and *resources*) are elements that can be internal to an actor or part of a dependency among two actors. In our approach, we further extend this to include *utilities* as an additional intentional element type. In our operational model, a goal is a functional requirement of the state of the affairs an actor eventually aims to *achieve*, for example assigning a car park space. On the other hand, a utility represents a non-functional requirement of the state of the affairs that is required to be *maintained* by an actor while achieving a goal. Examples of utilities are avoiding low battery levels or maintaining climate comfort while reaching the destination. Exemplary goals and utilities for the e-mobility case study are illustrated in Fig. 4.

B. SOTA Grammar and Language

Second, and since the i* model only addressed static aspects, goals and utilities are represented into an operational SOTA language, to describe the dynamic aspects of dependencies among service components. The syntax of such SOTA language adopts a context-free grammar, which consists of a number of productions or rules. To represent the dynamic characteristics, actors, entities and dependencies are represented as classes in the SOTA language (see Fig. 5). The grammar provides information on the class description defining the structure of the instances with their attributes. Attributes (constants or variables) of a SOTA class are used to represent relationships between objects. The satisfaction of goals and utilities can be expressed using three goal patterns: *achieve*, *maintain* and *avoid*. In general, a goal will have the *achieve* pattern while a utility will have the *maintain* or *avoid* pattern. The grammar also defines properties expressed in typed first-order linear-time temporal logic formula. Properties, which can be preconditions and postconditions, are used to describe the dynamic aspects of actors, entities and dependencies. Domain properties provide descriptive statements about the environment, which can be physical laws or organizational policies.

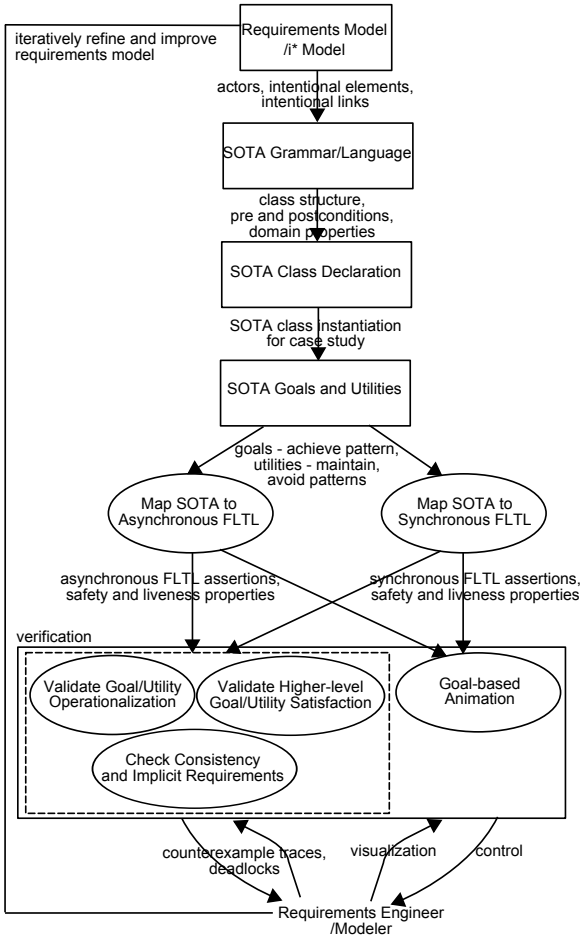

Figure 3: Model checking SOTA goal-oriented requirements

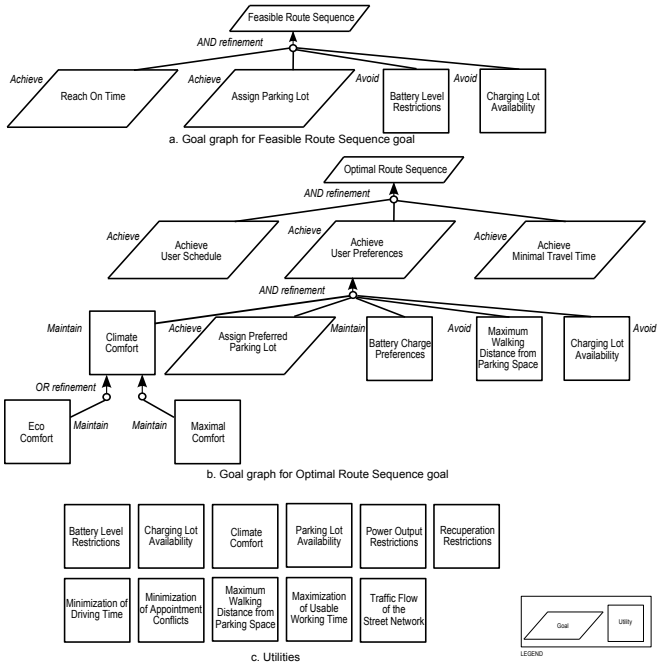

Figure 4: Goal graphs and utilities in e-mobility case study

```
Actor User
Actor Vehicle
    Attribute constant idNo: Integer
    Attribute variable soc: Integer
Dependency AssignParkingSpace
    DependencyType Goal
    GoalPattern Achieve
    Depender Vehicle
    Dependee ParkingLotOperator
    Attribute constant vehicle: Vehicle
    Attribute constant carpark: CarPark
    Precondition □¬(carpark.availableChargingLots = 0)
    Postcondition ◊(carpark.availableParkingSpaces > 0)
Dependency ChargingLotsAvailable
    DependencyType Utility
    GoalPattern Avoid
    Depender Vehicle
    Dependee ParkingLotOperator
    Attribute constant carpark: CarPark
    Postcondition □¬(carpark.availableChargingLots = 0)
Dependency ClimateComfort
    DependencyType Utility
    GoalPattern Maintain
    Depender User
    Dependee Vehicle
    Attribute constant vehicle: Vehicle
    Precondition vehicle.tripStart
    Postcondition □(vehicle.climateComfort='eco'
    ∨ vehicle.climateComfort='maximal')
Entity CarPark
    Attribute variable availableParkingSpaces: Integer
    Attribute variable availableChargingLots: Integer
```

Figure 5: Excerpt of a SOTA class declaration in e-mobility

C. Transform Goals and Utilities to Asynchronous FLTL

Third, a methodology to map SOTA to event-based, asynchronous FLTL of the LTSA for formal model checking purposes is provided. This mapping essentially allows the translation of SOTA goal-oriented requirements into an event-based model of service component behaviors, which is appropriate for reasoning and automated formal analysis at the software architectural level. To this end, each actor or entity, which can be a service component from the object model, is modeled as a finite state process in FSP. Then the bounding of the SOTA goal-oriented model is performed and fluents are identified. This involves transforming a SOTA goal-oriented model that has an infinite state space into a finite state fluents-based SOTA model. Next initiating and terminating events for each fluent are defined, and the preconditions and postconditions for goals and utilities are modeled as asynchronous FLTL assertions. In general, goals are modeled as liveness properties while utilities are modeled as safety properties in FLTL (see Fig. 7). Finally, the LTS model of the software-to-be is obtained by composing the component behavioral models with the FLTL assertions defined for the goals and utilities.

D. Verification

In the fourth stage of our model checking process, we verify the asynchronous FLTL assertions derived for the

goals and utilities. Model checking is applied to: validate whether a set of required preconditions and postconditions forms a complete operationalization of a single goal or utility; check the satisfaction of higher-level goals, which allows the engineer not to confine verification to a single goal or utility but a portion of the goal graph (i.e. multiple goal or utility operationalization); detect any inconsistencies and implicit requirements as deadlocks; perform animation of the goals-oriented models using the standard animation and simulation features of the model checker. The counterexample traces generated and the deadlocks detected in the model checking process are used to iteratively refine and improve the SOTA requirements model, thus deriving a specification that is correct.

V. Verification: an Example

Next we examine the actual verification of the model checking process using our running example from the e-mobility case study. The main objective of the example reported is to highlight some key advantages of the approach.

To illustrate the validating of a single goal or utility operationalization and higher-level goal or utility satisfaction, a goal and a utility from the running example is described next. In the description, first, the name of the goal or utility is provided with a keyword – *Achieve*, *Maintain* or *Avoid* – which specifies its temporal pattern. In general, the *Achieve* pattern is associated with goals while the *Maintain* and *Avoid* patterns with utilities. Next the goal or utility is described using natural language to facilitate communication with any stakeholder. This is followed by a formal definition of the goal or utility using first-order linear-time temporal logic notation. An LTSA model checker-based FSP and FLTL code is provided next to define the goal or utility for verification purposes. Finally, the results of model checking are discussed with any counterexample traces generated.

Utility Avoid[CHARGINGLOTSAVAILABLE]
Definition At least one charging lot needs to be available at the car park at all times.
Formal Definition $[]\neg$(carpark.availableChargingLots = 0)
The processes `VEHICLEARRIVALS` and `VEHICLEDEPARTURES` model the arrival and departure of vehicles to the car park (ln 7–8, Fig. 6). The `CHARGINGLOTCONTROL` process models a car park with two charging lots (ln 2–6). This process allows cars to enter the park when there is at least one charging lot. The `CARPARK` is the composed process of arrivals, departures, parking space control and charging lots control processes. `F_CHARGINGLOTSAVAILABLE` fluent is true if any of the initiating actions is true (i.e. there is no charging lot available). The utility `U_CHARGINGLOTSAVAILABLE` is an assertion which is true if and only if there is at least one charging lot available at the current instant and at all instants in the future. This assertion is violated when

```
1  const P = 2 // Number of charging lots in the car park
2  CHARGINGLOTCONTROL(P=2) = LOTS[P],
3  LOTS[i:0..P]            = (when(i>0) arrive->accessChargingLot->
4                             availableChargingLots[i-1]->LOTS[i-1]|
5                             when(i<P) depart->availableChargingLots[i+1]->
6                             LOTS[i+1]).
7  VEHICLEARRIVALS   = (arrive->VEHICLEARRIVALS).
8  VEHICLEDEPARTURES = (depart->VEHICLEDEPARTURES).
9  ||CARPARK = (VEHICLEARRIVALS||PARKINGSPACECONTROL(4)||CHARGINGLOTSCONTROL(2)||
10           VEHICLEDEPARTURES).
11 fluent F_CHARGINGLOTSAVAILABLE = <availableChargingLots[0],
12       {availableChargingLots[2],availableChargingLots[1]}>
13 assert U_CHARGINGLOTSAVAILABLE = []!(F_CHARGINGLOTSAVAILABLE)
```

Figure 6: FSP for CHARGINGLOTSAVAILABLE utility

```
1  const N=4 // number of car spaces in the car park
2  PARKINGSPACECONTROL(N=4) = SPACES[N],
3  SPACES[i:0..N] = (when(i>0) arrive->availableParkingSpaces[i-1]->SPACES[i-1]
4                   |when(i<N) depart->availableParkingSpaces[i+1]->SPACES[i+1]).
5  VEHICLEARRIVALS   = (arrive->VEHICLEARRIVALS).
6  VEHICLEDEPARTURES = (depart->VEHICLEDEPARTURES).
7  ||CARPARK = (VEHICLEARRIVALS||PARKINGSPACECONTROL(4)||CHARGINGLOTSCONTROL(2)
8               ||VEHICLEDEPARTURES).
9  assert G_POST_ASSIGNPARKINGSPACE = []<>availableParkingSpaces[i:1..N]
10 // Postcondition checks that whether a parking space will eventually be
11 // available. Precondition is the utility which checks whether there is
12 // a charging lot available
13 assert G_ASSNPARKSP_CHRGLOTAVAIL = (U_CHARGINGLOTSAVAILABLE
14                                                  && G_POST_ASSIGNPARKINGSPACE)
15 ...
16 // A higher-level goal with the use of the && operator
17 assert G_ASSNPARKSP_CLIMATECOMFT = (U_CLIMATECOMFORT
18                                                  && G_POST_ASSIGNPARKINGSPACE)
```

Figure 7: FSP for ASSIGNPARKINGSPACE goal

all charging lots are occupied in the car park. When the available charging lots is zero the fluent is false, thus violating the assertion. To this effect, a counterexample trace is generated, annotated with the names of fluents that are true when the action occurs.

Goal Achieve[ASSIGNPARKINGSPACE]
Definition Assigns a parking space to a vehicle. At least one charging lot needs to be available at the car park before checking whether a parking space will eventually be available.
Formal Definition $[]\neg$(carpark.availableChargingLots = 0) $\wedge$ $\Diamond$(carpark.availableParkingSpaces $>$ 0)
The `VEHICLEARRIVALS` and `VEHICLEDEPARTURES` processes model the arrival and departure of vehicles to the car park (ln 5–6, Fig. 7). The `PARKINGSPACECONTROL` process models a car park with four parking spaces (ln 2–4). It allows cars to enter the park when there is at least one space available, and allows a car to depart when there is at least one car in the park. The `CARPARK` is the composed process of arrivals, departures, parking space control and charging lots control processes.

The precondition for this goal is the `U_CHARGINGLOTSAVAILABLE` utility described previously, which specifies that at least one charging lot needs to be available at the car park. The postcondition

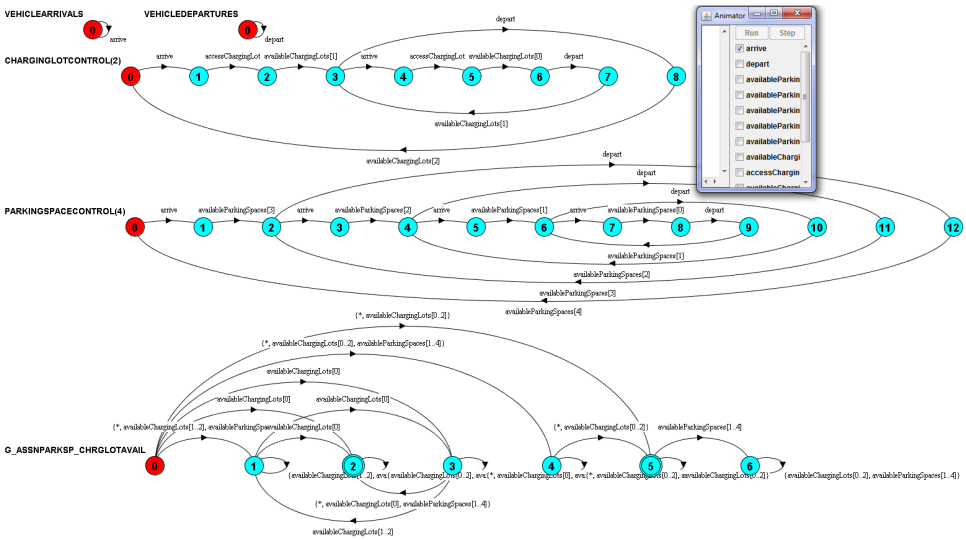

Figure 8: LTS model for the ASSIGNPARKINGSPACE goal

for this goal has been defined as a simple liveness property, which asserts that it is always the case that a parking space will eventually be available (G_POST_ASSIGNPARKINGSPACE, ln 9). The set of this pre and postconditions (ln 13–14) forms the complete operationalization of the ASSIGNPARKINGSPACE goal, which is an example of a *single goal operationalization*. The automatons generated are provided in Fig. 8. The LTS for G_ASSNPARKSP_CHRGLOTAVAIL corresponds to the negation of the property as required by model checking, which illustrates that * actions are required in state 0. These actions allow the automaton to non-deterministically move to the acceptance state. As there is parking space available there is no counterexample trace generated for the goal.

Ln 16–18 provide an example of a *multiple goal operationalization* with the use of the && operator. G_ASSNPARKSP_CLIMATECOMFT is a higher-level goal, which is concerned with achieving user preferences (see the User Preferences goal in Fig. 4). Here goal decomposition has been performed by refining it through the conjunction of U_CLIMATECOMFORT and G_POST_ASSIGNPARKINGSPACE.

The discussion so far, using our running example, exemplified the validating of a single goal or utility operationalization, and higher-level goal or utility satisfaction. Next we illustrate the checking for any inconsistencies and implicit requirements in the specification as deadlocks, and the goal-based animation performed on the models.

A typical problem that can occur in goal-oriented modeling is that an *inconsistency* or an *implicit precondition* can result in a deadlock in the specification. An inconsistency in the specification can occur for several reasons. For example, if the postcondition of a goal does not imply its precondition then the system might be in a state where the postcondition is true but the precondition is not. So the goal needs to be satisfied but it is not, leading to an inconsistency.

To illustrate this, let us consider the assign parking space goal discussed previously. A precondition was defined for this goal, which is the utility U_CHARGINGLOTSAVAILABLE (ln 10–14, Fig. 7). This utility specifies that at least one charging lot needs to be available at the car park. The precondition checks whether there is a charging lot available in the car park before considering availability of parking space. There can be a situation where there is a parking space available (postcondition satisfied) but no charging lot (precondition not satisfied). As a consequence, the goal needs to be satisfied but it is not, thus leading to an inconsistency (see counterexample trace in Fig. 9).

On the other hand, *implicit preconditions* occur due to interactions between requirements on different goals. An implicit precondition occurs in the following scenario. A postcondition of a goal may implicitly prevent another goal being applied even if all the preconditions for the second goal are true. Such a situation can cause a deadlock in the specification if we do not model additional constraints or properties to avoid it. Nevertheless, there is a benefit associated with implicit preconditions as it allows requirements engineers to eventually derive a robust specification.

The goal-oriented requirements models of SOTA can be *animated* using the standard animation and simulation features of the LTSA tool (for example, see Fig. 8). The gaol-based models can be explored interactively with stakeholders and error traces generated during formal analysis can be replayed, increasing the confidence in the validity of the goals and utilities.

The counterexample traces generated and the deadlocks detected in the model checking process (for example, see

Figure 9: Inconsistency detected in the ASSIGNPARKINGSPACE goal

Fig. 9) can be used to iteratively refine and improve the SOTA requirements model, thus deriving a requirements specification that is correct.

VI. Discussion and Evaluation

A discussion on our model checking approach, identifying some issues on scalability and state space, is provided here. Also, it provides some early ideas on evaluating the formal methods and tools used in our work.

A. Process Model, Scalability and State Space

All the steps of our model checking process, except the final step, are performed manually by the requirements engineer. The final step is the automated verification performed by the LTSA tool which can produce counterexample traces. The actual resource-consuming step of the model checking process is the translation of FLTL assertions to Büchi automaton [14] and their parallel composition. If a goal or utility has more than six fluents in its definition then it can be too large to be handled by the LTSA tool. However, such situations have not risen in the modeling of the case study, which is a non-trivial, real-world case study. Another scalability issue may arise if the engineer tries to compose all the component behavioral models with FLTL assertions. The intermediate state space generated during parallel composition may exceed the state space that can be handled by the LTSA tool. This is only an issue if the requirements engineer needs to obtain an explicit model of the parallel composition. However, such a representation is not required for model checking and animation purposes of the system. Instead the engineer can analyze a portion of the goal model rather that the entire system.

We could use an alternative approach to model checking SOTA with *synchronous* FLTL models of goals and utilities (map SOTA to synchronous FLTL, Fig. 3). This requires the event-based models to refer to explicit timing events, which can be modeled as tick events. However, the tick events make the synchronous model's state space much larger compared to the same system's untimed asynchronous model, which has no such tick events. This is a clear advantage of the asynchronous modeling of goal-oriented requirements, which is the approach adopted in the current study. Another

technique that can be used is mapping the goal-oriented models of SOTA to synchronous-based models used by the NuSMV model checker. However, such an approach will not integrate benefits of the two software engineering paradigms (i.e. requirements engineering and software architecture) in the same approach as in our work. Moreover, according to [17], it is a challenge to derive deadlock-free LTS models from goals defined using synchronous temporal logic, and specifying goals in asynchronous logic is the better approach. Thus, this justifies our approach of model checking asynchronous FLTL models of goals and utilities.

B. Evaluation: Formal Methods and Tools used

Clarke et al. [18] provide several criteria that formal methods-based approaches and tools need to support (i.e. early payback, incremental gain for incremental effort, multiple use, integrated use, ease of use, efficiency, ease of learning, orientation toward error detection, focused analysis and evolutionary development. In the present study, formal methods and tools (i.e. the LTSA tool) have been applied during the transformation and formal verification stages of the model checking process. We have evaluated our approach against all the criteria in [18]. Due to space limitations, we discuss two criteria here.

Early payback: this study is focused on the requirements engineering and architectural design levels of the software life-cycle. This goal-oriented requirements engineering and architecture-centric approach builds models of service components (i.e. behavioral model) with goals and utilities (i.e. functional and non-functional requirements that need to hold in the model), and verifies behavior against the specified system properties. Building goal-oriented and architectural models of complex self-adaptive systems allows the engineers to validate their actual correctness before implementation later in the software life-cycle. Thus, it provides early payback or feedback to the software engineer on the validity of these complex systems.

Incremental gain for incremental effort: each stage of this model checking process has its own deliverables, such as the requirements model using i*, the SOTA language, the behavioral model with asynchronous FLTL assertions for goals and utilities, and the model checked specification. This demonstrates that the engineer can receive benefits of the process in an incremental manner.

VII. Related Work

There are several works in the literature that are related to the current research.

The SOTA language of our operational model was motivated by the formal Tropos language [19]. Like formal Tropos, we have represented actors, entities and dependencies from the i* framework as classes. Also, properties have been expressed in typed first-order linear-time temporal logic formula. However, we introduce the notion of *utilities* as an additional intentional element type. Another distinction of our work is it is based on the self-adaptive systems domain. We introduce the goal patterns – *Achieve*, *Maintain* and *Avoid* – in the language to specify the temporal pattern of a goal or utility. The notions of *precondition* and *postcondition* for goals and utilities are explicitly specified in our work. Moreover, our model checking approach is based on the LTSA and not the NuSMV model checker.

The most widely used goal-modeling notations, such as the i* framework, the KAOS methodology and the Tropos methodology, provide no explicit support for handling uncertainty or adaptivity. As stressed in [20], the current languages for requirements engineering do not handle uncertainty, which is a key consideration for self-adaptive systems. As a result, several researchers have proposed extensions to these goal-modeling notations. For example, in [9] the authors present an approach for embedded self-adaptive systems, which uses the strengths of the i* modeling framework. Tang et al. [6] discuss a systematic approach for the goal-directed modeling of self-adaptive software architecture. It systematically derives a self-adaptive software architecture model from a KAOS-based requirements goal model. In another approach, Yu et al. [5] advocate a systematic process for generating complementary design views from a goal model while preserving variability. The process is supported by heuristic rules and mapping patterns. In [10], the authors discuss a Tropos-based framework, focussing on the requirements engineering perspective for self-adaptive systems. The Tropos methodology has been incorporated with the REFLEX language specification constructs to support uncertainty issues in self-adaptive systems environment. In another Tropos methodology-based extension called Tropos4AS (Tropos for Adaptive Systems) [4], it allows the engineer to capture and detail during design time, any specific knowledge and decision criteria needed to guide self-adaptation at run-time. In our work, we express variability of the adaptation requirements using the notions of goals and utilities.

In summary, there are several works on the goal-oriented requirements engineering of self-adaptive systems as discussed above. However, these works provide very little support for automated verification analysis techniques such as formal model checking. Assuring correctness in complex self-adaptive systems is essential, which consists of functional and non-functional properties. Thus, providing motivation for the current research.

VIII. Conclusions and Future Work

The primary contribution of this paper is a novel, systematic, incremental approach for model checking complex self-adaptive systems, which essentially integrates and leverages the benefits of goal-oriented requirements elaboration with formal analysis techniques on event-based systems. In particular, our model checking approach is based on an

operational, event-based, untimed and asynchronous model of labeled transition systems. This work has shown that the formal model checking of goals and utilities, as derived from the operationalization of the SOTA modeling, can help to iteratively refine and improve the SOTA requirements model, building a requirements specification that is correct. The approach has been explored and validated using the e-mobility case study of the ASCENS project, adapted to the goal-oriented requirements modeling domain.

As for future work, we will be investigating on how to use our SOTA conceptual model to derive the knowledge and awareness requirements of the system, and to help understand according to which architectural scheme a system should be architected so goals can be adaptively achieved.

ACKNOWLEDGMENT

This work is supported by the ASCENS project (EU FP7-FET, Contract No. 257414).

REFERENCES

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[2] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software Engineering for Self-Adaptive Systems*. Springer, 2009, pp. 48–70.

[3] F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani, "On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles," in *Proceedings of the 1st Awareness Workshop at the 5th IEEE International Conference on Self-adaptive and Self-organizing Systems*, Ann Arbor (MC), Oct. 2011.

[4] M. Morandini, L. Penserini, and A. Perini, "Modelling self-adaptivity: A goal-oriented approach," in *Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2008, pp. 469–470.

[5] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. P. Leite, "From goals to high-variability software design," in *Proceedings of the 17th International Conference on Foundations of Intelligent Systems*. Springer, 2008, pp. 1–16.

[6] S. Tang, X. Peng, Y. Yu, and W. Zhao, "Goal-directed modeling of self-adaptive software architecture," in *Proceedings of the 14th International Conference on EMMSAD'09*. Springer, 2009, pp. 313–325.

[7] K. Rasch, F. Li, S. Sehic, R. Ayani, and S. Dustdar, "Context-driven personalized service discovery in pervasive environments," *World Wide Web*, vol. 14, no. 4, pp. 295–319, Jul. 2011, springer, Netherlands.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, UK: The MIT Press, Dec. 1999.

[9] P. Sawyer, N. Bencomo, D. Hughes, P. Grace, H. J. Goldsby, and B. H. C. Cheng, "Visualizing the analysis of dynamically adaptive systems using i* and DSLs," in *Proceedings of the 2nd International Workshop on Requirements Engineering Visualization*. IEEE Computer Society, 2007, pp. 1–10.

[10] F. Noman and Z. Nasir, "A Tropos based requirement engineering frameworks for self adaptive systems," *International Journal of Computer Science and Information Security*, vol. 8, no. 4, pp. 60–67, Jul. 2010.

[11] N. Hoch, B. Werther, H. P. Bensler, N. Masuch, M. Ltzenberger, A. Heler, S. Albayrak, and R. Y. Siegwart, "A user-centric approach for efficient daily mobility planning in e-vehicle infrastructure networks," in *Advanced Microsystems for Automotive Applications 2011*, ser. VDI-Buch, G. Meyer and J. Valldorf, Eds. Springer-Verlag, 2011, pp. 185–198.

[12] M. Morandini, L. Sabatucci, A. Siena, J. Mylopoulos, L. Penserini, A. Perini, and A. Susi, "On the use of the goal-oriented paradigm for system design and law compliance reasoning," in *Proceedings of the 4th International i* Workshop (i* 2010) at CAiSE'10*, Jun. 2010, pp. 71–75.

[13] A. K. Dey and G. D. Abowd, "Towards a better understanding of context and context-awareness," in *Proceedings of the CHI 2000 Workshop on The What, Who, Where, When, Why and How of Context-Awareness*, Apr. 3, 2000, ftp://ftp.cc.gatech.edu/pub/gvu/tr/1999/99-22.pdf (Last accessed on 25/10/2011).

[14] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd ed. John Wiley and Sons, Apr. 2006.

[15] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *Proceedings of the 9th European Software Engineering Conference*. ACM, 2003, pp. 257–266.

[16] E. S. K. Yu, "Towards modeling and reasoning support for early-phase requirements engineering," in *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*. IEEE Computer Society, 1997, pp. 226–235.

[17] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Deriving event-based transition systems from goal-oriented requirements models," *Automated Software Engineering*, vol. 15, no. 2, pp. 175–206, Jun. 2008.

[18] E. M. Clarke, J. M. Wing, and R. Alur, "Formal methods: State of the art and future directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, Dec. 1996, ACM.

[19] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso, "Specifying and analyzing early requirements in Tropos," *Requirements Engineering*, vol. 9, no. 2, pp. 132–150, May 2004, springer-Verlag, Secaucus, NJ, USA.

[20] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, "Software Engineering for Self-Adaptive Systems." Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.