How Software Changes the World: The Role of Assumptions

Xiaowei Wang and John Mylopoulos
Department of Information Engineering and Computer Science
University of Trento, Italy

Giancarlo Guizzardi
Ontology and Conceptual Modeling Research Group
Federal University of Espírito Santo, Brazil
ISTC-CNR Laboratory for Applied Ontology, Trento, Italy

Nicola Guarino
ISTC-CNR Laboratory for Applied Ontology, Trento, Italy

***Abstract*—The requirements for most software systems -- the intended states-of-affairs these systems are supposed to bring about -- concern their operational environment, often a socio-physical world. But software systems usually don't have any direct means to change that environment in order to bring about the intended states-of-affairs. In what sense then can we say that such systems fulfill their requirements? The main purpose of this paper is to account for this paradox. We do so by proposing a preliminary Ontology of Assumptions. This ontology aims to characterize and make explicit a number of notions that are used implicitly in software engineering practice to establish that a system specification S fulfills its requirements R given a set of assumptions A. Our proposal is illustrated with an example concerning a meeting scheduler.**

Keywords—software requirements; software specifications; domain assumptions

I. INTRODUCTION

Consider a software system that schedules meetings upon request. Its ultimate requirement is not only to produce some information, consisting of a schedule that satisfies some given constraints, but also to bring about a change in the social world where the software operates, and ensure that the proper actions necessary for meeting organization (such as invitations to participants and room allocation) are effectively undertaken in terms of software outputs. The actual effectiveness of such a software system will be evaluated on the basis of its impact on the social environment, i.e., does it schedule meetings, or not? However, a software system consisting of software and various user interfaces (let's call it "pure software system") by its very nature, can only change the states of the machine within which it operates.

There seems to be a paradox here. The requirements for most software systems, the intended states-of-affairs these systems are supposed to bring about, concern their operational environment, usually a socio-physical one. But these systems can only affect machine states, not the socio-physical ones. We call this the *world-on-machine paradox*. It is important to emphasize that we are talking about pure software systems that interact with users through interfaces, yet they have no means to change the physical and/or social world within which they operate. In other words, this kind of system only consist of software and user interfaces, and doesn't include robotic or other components that can manipulate their socio-physical operational environment. In what sense then can we say that such systems fulfill their requirements?

The main purpose of this paper is to account for the world-on-machine paradox. We do so by proposing an ontology of assumptions that are implicitly used in software engineering practice to establish that a system specification *S* fulfills its requirements *R* given a set of assumptions *A*. Adopting the formula of requirements problem initially proposed by Jackson and Zave [1], our task is to characterize the nature of the assumptions used and needed to establish that

$$A, S \models R \tag{1}$$

given that the requirements are about world phenomena (e.g., meetings, participants, timetables, rooms, and etc.), while the specification is about machine phenomena (database tables, tuples) and manipulations thereof.

Several researchers (e.g., Lewis [2], Abdullah and et al. [3], Brown [4], Tun and et al. [5]) have emphasized the importance of assumptions, and have proposed techniques for capturing them. Our proposal goes further in the sense that it identifies new kinds of assumptions (notably, world and machine dependence ones) that have not been previously accounted for. The specific contributions of this paper are three-fold:

1) Proposes a preliminary ontology of assumptions, consisting of four kinds of assumptions. Two of them are based on literature work, including *world assumptions* and *machine assumptions*. Two new kinds of assumptions are proposed additionally, including *world dependence* and *machine dependence assumptions*. We claim that these four kinds of assumptions together define a solution to the paradox mentioned earlier, and elaborate on the important role they play in linking world and machine states;

2) Clarifies the concept of 'assumption', identifying two possible interpretations that are both important for software engineering, namely *assumptions-used* and *assumptions-*

needed; on the basis of this distinction, the paper provides an update to Jackson and Zave's original formulation of the requirements problem;

3) Discusses how our results can be employed methodologically, suggesting how software developers should systematically and explicitly manage all four kinds of assumptions proposed here. We suggest that these assumptions should be explicitly identified and systematically guaranteed to hold throughout the useful lifetime of their software systems.

The rest of the paper is structured as follows. In section 2, we introduce our research baseline, consisting of the Jackson and Zave's formulation of the requirements problem [1], as well as the *situation calculus*, originally proposed by McCarthy [6], which we use to model assumptions, specifications and requirements. In section 3, we analyze the link between the world and a software-driven machine by defining a clear boundary between them, classifying and representing concerned phenomena according to this boundary. In section 4, we propose a preliminary ontology of assumptions, illustrating the four kinds of assumptions that enable the link between world and machine. Also, in this section, we explain our interpretation of the role of assumptions in the formula '$A, S \vDash R$', and we argue for the importance of making a precise interpretation of such assumptions explicit. In section 5, we make use of the situation calculus to model a meeting scheduler example that elaborates on the role of assumptions in establishing the link between world and machine states. Finally, section 6 discusses related work, while section 7 concludes and discusses future research questions.

II. BASELINE

Requirements, Assumptions, and Specifications

The requirements problem was formulated as '$A, S \vDash R$'[1] by Jackson and Zave in 1995, where R, S and A refer respectively to requirements, specification, and assumptions. The problem itself consists of finding an S for given R and A, such that '$A, S \vDash R$'. In other words, the satisfaction of the requirements is entailed by the specification together with the assumptions.

Later on, Jackson and Zave reinterpreted the formula, and especially emphasized the role of assumptions in ensuring the satisfaction of the formula [7]. They pointed out that the gap between requirements and specification had long been recognized, and also, it had long been recognized that domain assumptions about the environment should play an important role in requirements engineering. However, they did not fully elaborate neither on the nature and purpose of assumptions, nor on how assumptions should be identified and modelled.

After diving into the four dark corners of requirements engineering [7], Jackson and Zave proposed that it was the domain assumptions that bridge this aforementioned gap between requirements and specification. In particular, they defended the thesis that requirements can always be satisfied by a specification with the help of suitable domain assumptions. In other words, whatever is missing to establish the entailment of requirements from specification should be captured in terms of assumptions. We concur in this paper with this view. However, we take a further step to explain in details what kinds of assumptions are used and needed to ensure the satisfaction of the formula, as well as the role of these assumptions in linking world and machine states.

[1] This formula was initially '$S, E \vdash R$', but here it is amended, on the basis of subsequent work.

The Situation Calculus

We choose the situation calculus as our representation language for requirements, assumptions and specifications. The situation calculus is a logic formalism proposed by McCarthy [6] in 1963 to represent and reason about dynamic domains. It is chosen here because it is a well-known language, used in artificial intelligence to capture state changes. This makes it quite suitable for demonstrating links between world and machine state changes. We highlight however that the situation calculus is used here for illustration purposes, and we don't expect that it will be used by software engineers who need to capture assumptions for their system.

A *situation s* constitutes a complete state of affairs at some instant of time. In other words, it is a snapshot of the world described in terms of properties, *fluents*, that hold or don't hold. A fluent (e.g., $meeting\_scheduled_W(Mtg, S_n)$) is usually represented by a predicate (e.g., $meeting\_scheduled_W$) having zero or more arguments, and a situation (e.g., S_n) as its final argument.

An *action* (*a*) causes a transition from one situation to another, and causes changes to the truth value of fluents. Similarly to *function* in software engineering, an action may have a *pre-condition* described by precondition axioms and a *post-condition* described by effect axioms. The former indicate the situations where the action can be executed; the later indicates the changes in truth values of fluents after the execution of the action.

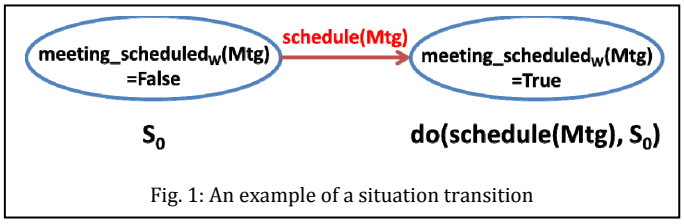

Fig. 1: An example of a situation transition

The new situation achieved by executing an action (*a*) in a certain situation (*s*) is denoted by '*do(a, s)*'. Fig. 1 shows what happens when an action is executed in a certain situation, resulting in a new situation. In this example, in the initial situation S_0, the value of the fluent $meeting\_scheduled_W(Mtg)$ is *False*. By executing the action *schedule(Mtg)* in that situation, a new situation is achieved denoted by $do(schedule(Mtg), S_0)$. In this new situation, the value of the fluent $meeting\_scheduled_W(Mtg)$ becomes *True*, and this can be represented as $meeting\_scheduled_W(Mtg, do(schedule(Mtg), S_0))$.

III. FROM MACHINES TO WORLDS

As argued in our previous work [8], [9], a software program possesses a peculiar characteristic when compared with other kinds of information artifacts (e.g. recipes or laws): it plays the role of a bridge between the symbols in a machine and the phenomena in its outside world. More specifically, while other

kinds of information artifacts directly refer to the objects in the world (so that executing a recipe or a law implies a manipulation of objects in the world), software programs refer to variables and states in a machine, whose manipulation inside the machine affects the outside world in an indirect way.

When a software program is embedded in a machine to control its external behaviors, we have a *software-driven machine*. The ultimate purpose of a software-driven machine program is to constrain the phenomena of its external environment. The machine monitors and controls the environment by means of *transducers* bridging symbolic data and physical properties of the environment (hereafter 'software-driven machine', or simply 'machine' when the meaning is clear in context).

In the case of a stand-alone personal computer (PC) such transducers just concern the human-computer interface and the standard I/O devices; for mobile systems they may also include position and acceleration sensors, while in the case of embedded systems they take the form of ad-hoc physical sensors and actuators. So, in the general case, a software system's ultimate purpose is achieved by running a program that produces certain effects inside a computer, which drives a physical machine, which in turn produces physical effects on its external environment.

However, we may also wonder whether software-driven machines can affect their social environment. Indeed, as we have seen in the case of the meeting scheduler example, in many cases the ultimate purpose of software is to produce such changes in the social world. But how is this possible, given that the machine has no direct means to manipulate or otherwise affect the social world? This is the paradox we introduced at the beginning of this paper.

To address this question, let us first consider a different scenario that plays out in the social world, and is affected by artifacts other than software-driven machines. In modern monetary systems, certain kinds of colored, marked paper is used as money in business transactions. As we all know, giving a piece of such paper to a person has important social effects concerning your and his/her net worth [10]. This happens because we share the same *assumption* about the ownership of such pieces of paper, and the net worth of social actors.

The *count-as* relationship was proposed by John Searle to link what he terms a *brute physical fact* (e.g., presenting a bill) with what he terms an *institutional fact* (e.g., having the right to obtain a good in exchange of the bill). In another example, two hands joined in a handshake count as an institutional relationship (e.g., an agreement being created in a given social context). We would say therefore that the agreement *depends* on the handshake [11] under the assumption that handshakes count as agreements.

Moving now to software-driven machines, we note that the monetary system has evolved with the advent of computers: data in a bank's database counts as money too! In other words, our business world is actually controlled by software-driven bank machines, which literally have the power to change the world, thanks to *count-as* assumptions [10].

Going back to the meeting scheduler, we have another case where machine-based facts *count-as* institutional facts, since a meeting record marked as scheduled in the computer *counts-as* the collective belief of meeting participants that the meeting is actually scheduled, with corresponding commitments by all concerned.

Starting with the formula '$A, S \vDash R$', Jackson et al. have proposed in a series of papers ([12], [7], [13], [14]) to pay attention to the boundary between world and machine, drawing a clear distinction between the social environment where the ultimate effects of a software system are expected, and the machine where the system operates. Lamsweerde recognizes their work in a paper entitled 'from worlds to machines' [15], which elaborates on how a specification concerning the behaviors of a machine could be derived from a set of requirements concerning its external world. In this paper we take a complementary perspective, focusing on how software-driven machines can affect the world. In other words, given a specification and a set of requirements, we explain in what sense the requirements can be entailed from the specification.

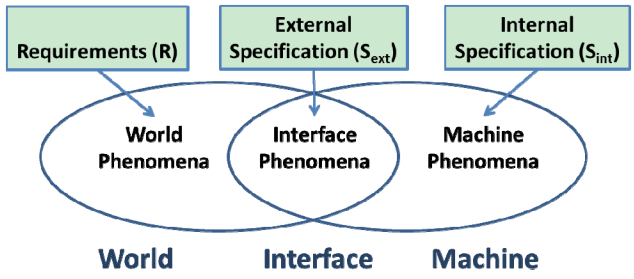

Fig. 2: Cutting the boundary between World and Machine

As shown in Fig. 2, according to Jackson and Zave, in a software engineering scenario, the phenomena of interest could be classified into three categories according to their controllability and visibility, namely: *world phenomena*, which can only be seen and controlled by the world; *interface phenomena*, which can be seen both by the world and the machine, and can be controlled either by the world or the machine; and *machine phenomena*, which can only be seen and controlled by the machine.

In [8] and [9], we proposed the term 'internal specification' to refer to a specification that only constrains the phenomena happening inside the machine; we distinguish it from the 'external specification' (originally called 'specification' in the work of Jackson et al.), which constrains the phenomena happening at the interface.

To capture the relationships between the phenomena in these different layers, we firstly capture and represent these concerned phenomena in a preparatory step. Hence, in the remainder of this section, we elaborate on how a formalism such as situation calculus could be used to represent these requirements, as well as the external specification and the internal specification in the sense put forth by the aforementioned classification of different kinds of phenomena.

Requirements (R)

We interpret a requirement as a set of (conditional) states of affairs that are intended by stakeholders. Based on the introduction of situation calculus stated in the baseline, we can represent a set of states of affairs as a set of situations. In an intended situation, the concerned world fluents have the specific values intended by the stakeholders. If the requirement has a conditional nature, a situation transition will be specified. For example, a requirement may be 'a meeting shall be scheduled after a meeting initiator intends to schedule one', meaning that if we start in a situation where a meeting has been intended by an initiator, some actions will get us to a situation where the intended meeting is scheduled. Following this view, we can represent this requirement in a situation calculus formula as follows:

$$R_0 : \exists a\{Poss(a,s) \leftrightarrow meeting\_intended_W(initiator, meeting, s) \wedge Poss(a,s) \rightarrow [meeting\_scheduled_W(meeting, s') \wedge s' = do(a,s)]\}$$

This formula means that there exists an action *a* such that, starting from an initial situation *s* where a *meeting* is intended by an *initiator*, action *a* can bring about a new situation where the meeting is scheduled. The action *a* here in the formula refers to an action variable in a second order logic. Also, note that we use subscripts *w (world), i (interface),* and *m (machine)* to distinguish among the fluents concerning the different kinds of phenomena discussed in Fig. 2.

In a software engineering process, such an action could be understood as an alias of a function, and this decision is adopted by many software engineering standards, such as IEEE-STD-830-1993. Hence a set of functions defined in the specifications becomes a solution to a requirements problem. As previously stated, according to the world-machine distinction, we classify specifications into external specifications and internal specifications respectively, and here we introduce each of them as follows[2]:

External Specification (S_{ext})

An external specification contains a set of actions $S_{ext} = \{a_{I1}, a_{I2}, \ldots, a_{In}\}$, which are supposed to occur at the interface in order to satisfy the requirements. If the specification is implemented correctly, executing an interface action will bring about a intended situation transition, resulting in changes of the interface fluents, and that (by the very definition of interface) will be visible both from the machine and the outside world.

Besides that, it is important to highlight that, an action in S_{ext} may be applied several times to get the stakeholders to an intended situation, and here the detailed executing order of this series of actions is ignored. For example, for a meeting scheduling system, if S_{ext} includes an interface action 'receive timetable from a participant', this action will have to be applied many times to get the timetables from all the participants for a meeting, so that the stakeholders can reach the situation where all timetables have been collected.

Internal Specification (S_{int})

An internal specification contains a set of machine actions $S_{int} = \{a_{M1}, a_{M2}, \ldots, a_{Mn}\}$, concerning machine phenomena. Executing a machine action will bring about a situation transition, resulting in machine fluent changes. Although the user of a software system may not be [3] interested in implementation details, software engineers certainly are, and they need to provide machine actions supplementing interface actions. Together, external and internal specifications, provide a complete solution to a given set of requirements. As with external specifications, actions in an internal specification are introduced without any details about their behavior, i.e., execution order.

IV. A Preliminary Ontology of Assumptions

A key objective of this paper is to identify and characterize different kinds of assumptions, notably the newly proposed kinds of dependence assumptions. We explain the key role of these assumptions in determining the relationship between the social world and a software-driven machine, thereby addressing the paradox mentioned earlier. Besides that, we introduce the interpretations of the term 'assumption' adopted in the software engineering community as 'assumption-needed' and 'assumption-used' in our ontology of assumptions. As demonstrated in the end of the section, these interpretations are corroborated by literature in the legal domain. We emphasize that this ontology is a partial attempt in systematizing the related notions. In a future paper, we expect to elaborate on the detailed ontological nature of these assumptions, in particular, in aspects dealing with the mental states/attitudes.

A Classification of Assumptions

As shown in Fig. 3, four kinds of assumptions are included in our proposal: world assumptions (*WA*), machine assumptions (*MA*), world dependence assumptions (*WDA*), and machine dependence assumptions (*MDA*). We introduce each of them in the sequel.

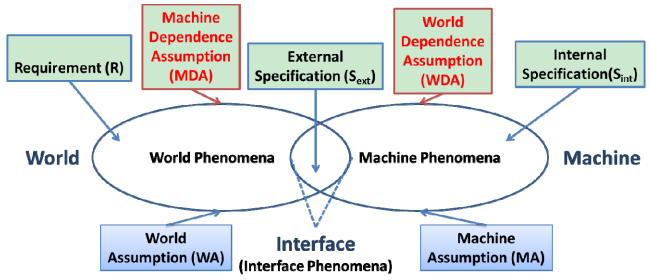

Fig. 3: An extension of the work in cutting the boundary between worlds and machines

A *world assumption* is an assumption about world phenomena, not visible by the machine. Such assumptions constrain the environment of the software-driven machine. For example, for a meeting scheduling system, we may assume that

[2] Note that actions mentioned here are actually at the type/class level, while action instances represent action executions.

[3] Put more precisely, almost surely …

there are rooms available for all requested meetings. Such an assumption means that our solution does not work when no room is available for some meeting requests (e.g., during a busy period with many meeting requests).

A *machine assumption* is an assumption about a machine's internal phenomena, i.e., those that are only visible to the machine. For instance, an action that records a meeting by adding it to a table requires the assumption that there is always space available on the table for all recorded meetings. Such assumptions often simply ensure the availability of computational resources within the machine. For another example, we need to assume that there is power for the machine to run, meaning that if this assumption fails, meeting scheduling may not work.

Since world and machine assumptions are either about the world (e.g., the assumption of enough rooms for the meetings) or about the machine (e.g., the assumption of the power supply for the machine) independently, they are not sufficient to describe the causal connection between machine and world states, and can't answer the question in the very title of this paper: how can the machine change the world? To answer this question we need to focus on the interface between world and machine and consider another two kinds of assumptions: *world dependence assumptions* and *machine dependence assumptions*.

These two additional kinds of assumptions are proposed specifically to constrain the relationship between world and machine phenomena. For example, the value reported by a sensor is within 2 degrees of room temperature. Here, machine phenomena depend on world ones. Conversely, the action undertaken by an actuator depends on the value of a machine variable. Here, world phenomena come about/caused by machine ones.

A *machine dependence assumption* states that an external world phenomenon depends on some machine phenomena. For instance, we may assume that a certain meeting has been scheduled in the world once an entry has been added to the meetings table with particulars for the meeting, a room is reserved for a certain meeting at a certain time (and therefore it will not be used for any other purpose at that time).

In contrast, a *world dependence assumption* states that a machine phenomenon depends on some world phenomena. For example, we may assume that whenever a certain room appears to be free on the machine, it is because the room is actually free in the external world. Similarly, we can assume that whenever a meeting appears to be requested in the machine, it is because somebody actually intended to schedule such meeting. In other words, a world dependence assumption is an assumption about the correspondence between states of the machine and the phenomena in the world these states are supposed to represent. The closed world assumption of Raymond Reiter [16] constitutes an early example of a world dependence assumption.

In the case of a machine dependence assumption, the depending phenomenon in the world could be physical or social. When it is physical, it means that there is a path of physical interactions connecting an observed phenomenon in the external world with machine phenomena. This is the common scenario in cyber-physical systems, which interact with the external world by means of actuators and sensors. However, our primary interest here is with social phenomena depending on machine ones. In the following, we shall explore the nature of this dependence, thereby accounting for the world-on-machine paradox.

The Causal Chain Enabled by these Assumptions

Four kinds of assumptions were proposed in the preceding subsection. We now explain the key role played by these assumptions in linking the world and machine states. In particular, we introduce a causal chain enabled by these assumptions: some triggering phenomenon occurs in the outside world (a meeting request), it propagates through the interface, and reaches the symbolic states inside the machine (get timetables, select a timeslot); then the chain returns to the outside world by crossing back through the interface (participants are informed, meeting has been officially scheduled). Once again, we rely on the meeting scheduling example to explain the causal role of these assumptions.

An instance of a whole meeting scheduling process starts from the initial state S_0 in which a meeting initiator intends to schedule a meeting, which could be represented as a world fluent $meeting\_intended_W(Initiator, Mtg, S_0)$. From this state, the initiator instigates action a_{I0} provided by the interface to enter the meeting information through the interface, and this is represented by the action's pre-condition that $PRE: WDA_0: meeting\_intended_W(Initiator, Mtg, S_0)$, and post-condition $POS: meeting\_information\_entered_I(Mtg, do(a_{I0}, S_0))$, and this step can be summarized as a situation transition enabled by the interface action a_{I0} formalized as follows:

$$a_{I0} : Enter\_Meeting\_Information(initiator, meeting)$$

$$PRE : WDA_0 : Poss(a_{I0}, s) \leftrightarrow meeting\_intended_W(initiator, meeting, s)$$

$$POS : Poss(a_{I0}, s) \rightarrow [meeting\_information\_entered_I(meeting, s') \wedge s' = do(a_{I0}, s)]$$

The world dependence assumption WDA_0, asserted in the pre-condition of the interface action a_{I0}, is meant to capture the relation between a state of the machine (representing the possibility of the execution of action a_{I0}) and a state of the world (the mental attitude, i.e., the intention of a given human agent). Without this assumption, there is no means to constrain the execution of the action a_{I0} according to a state of the human mind (represented here by $meeting\_intended_W(initiator, meeting, s)$). In other words, the situation transition from s to s' enabled by the action a_{I0} won't work without WDA_0 being somehow ensured. This transition can be visualized by the single direction arrow between S_0 and S_1 in Fig. 4.

When the meeting information is entered, the machine can sense this change in $meeting\_information\_entered_I(meeting)$, and then execute the machine function a_{M0} to receive the information from the interface, insert a corresponding meeting record into the database, resulting in the change in the machine fluent $meeting\_record\_inserted_M(meeting)$. The action a_{M0} can be formalized as follows:

a_{M0} : *Insert_Meeting_Record(meeting)*

$PRE : WDA_1 : Poss(a_{M0}, s) \leftrightarrow$

$meeting\_information\_entered_1(meeting, s)$

$POS : Poss(a_{M0}, s) \rightarrow$

$[meeting\_record\_inserted_M(meeting, s')$

$\wedge s' = do(a_{M0}, s)]$

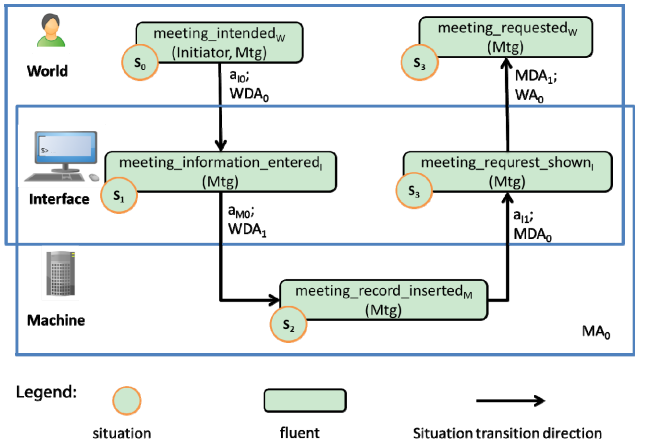

Fig. 4: The causal chain underlying software engineering enabled by assumptions

The action a_{M0}'s pre-condition follows a_{I0}'s post-condition, and this ensures the continuity of the situation transitions from S_0 to S_1 and then to S_2. No surprise that another world dependence assumption WDA_1 is made, and asserted into the pre-condition of the action a_{M0}. This assumption indicates that the execution of the action depends on the physical framework provided by the interface. In other words, when this assumption is fulfilled, the machine could sense the interface fluent and then execute the corresponding action a_{M0}.

Till now, the route of the changes in fluents has travelled from the world to the machine through the interface. Now, it is time to consider how it can travel back. To achieve that, we need to reach the interface first, and another interface action a_{I1} is introduced. By sensing the value of the machine fluent $meeting\_record\_inserted_M\ (meeting)$ through the machine dependence assumption MDA_0, the action a_{I1} will be executed resulting in a change at the interface fluent $meeting\_request\_shown_I(message)$ indicating the message of 'meeting is successfully requested' is shown on the screen. As before, the action a_{I1} can be represented as follows, yet note that a machine assumption MA_0 is also inserted here to ensure the power supply for the whole process of the situation transitions.

a_{I1} : *Show_Meeting_Request(meeting)*

$PRE : MDA_0 : Poss(a_{I1}, s) \leftrightarrow$

$meeting\_record\_inserted_M(meeting, s)$

$POS : Poss(a_{I1}, s) \rightarrow$

$[meeting\_request\_shown_I(meeting, s')$

$\wedge s' = do(a_{I1}, s)]$

MA_0 : $power\_supply_M(Machine)$

Now, we reach the interface successfully, and there is only one step left to travel back to the outside world. To fill this gap, we make two additional assumptions: 1) MDA_1 indicates that as soon as the message is shown on the screen at the interface, we assume that the stakeholders all will agree with that the meeting is requested; 2) WA_0 constrains the system that it is designed for people who are not visually impaired.

$MDA_1 : meeting\_requested\_message\_shown_I(message, s)$

$\leftrightarrow meeting\_requested_W(meeting, s)$

$WA_0 : can\_see_W(people)$

A brief demonstration of the role of assumptions in the software engineering process is presented in this section. As one can see, fluents change through situation transitions (e.g., following the sequence of S_0, S_1, S_2, and S_3). The chain travels from the world, crosses the interface, reaches the inside machine, then returns to the world crossing the interface again. This causal chain is only possible thanks to the assumptions introduced in this paper that link world and machine phenomena together.

Interpretations of the Concept of Assumption[4]

'Assumption' is a severely overloaded term used in many communities (e.g., research, industry, and etc.) as well as in our daily lives. The interpretations of this term diverge significantly in different contexts: in the sequel, we present a few examples of these possible interpretations (relying on the language use of the term) [17]:

Conclusion: e.g., Tom said: "my assumption is that you are going out, since you are wearing your cap." The conclusion of 'going out' is derived from the current situation 'wearing your cap'.

Less-than-fully established proposition, in an accusation sense: e.g., Mike answered: "that is only your assumption, you don't know it." Mike replied that it might look like he's going out, yet that was only Tom's guess and as such it is not guaranteed to hold.

Adopted in order to deceive, fictitious, pretended: e.g., "although bad things happened, please assume that they didn't ever happen." The term assumption is interpreted as a kind of 'self-deception' here that 'you can deceive yourself that nothing bad happened'.

The examples aforementioned are only a small part of a full possible list. However, considering the importance of the role played by assumptions in software engineering process, it is necessary for stakeholders to achieve an agreement on the interpretation of this term. Fortunately, a clarification of this

[4] The interpretations of assumptions mentioned in this subsection are orthogonal with the preceding four kinds of assumptions.

term was proposed by Ennis in 1982, providing clear guidance in interpreting this term according to its use in practice [17]. More precisely, he classified assumptions into two main kinds, namely '*assumptions-used*' and '*assumptions-needed*'. Assumptions-used are propositions that a person uses *a priori* while constructing a new argument. Scientific theories are examples of such assumptions, adopted by scientists as the foundational components of a theory (e.g., the law of gravity). On the other side, assumptions-needed are propositions that are needed *a posteriori* to support a previous conclusion. In this sense, they circumscribe the contexts within which the conclusions are reasonable.

In what follows, we adopt this distinction between assumptions-used (*AU*) and assumptions-needed (*AN*). We use the distinction to elaborate on the aforementioned Jackson and Zave's formulation of the requirements problem. In particular, given a set of requirements *R* and a set of assumptions-used *AU*, one needs to find a specification *S* and a set of assumptions-needed *AN* such that $AU, AN, S \vDash R$. To present a simple example, suppose one is given a requirement *R*: 'Fly to the moon' and *AU*: 'laws of gravity'. An engineer then needs to find a specification for a spacecraft *S* that will fulfill *R* provided that the following assumption holds, *AN*: 'spacecraft carries enough fuel'.

The choice of interpreting assumptions as assumptions-used or assumptions-needed has strong practical implications that can be illustrated by legal disputes between product providers and users. If a product malfunctions because of an assumption-used, the malfunction is the problem of the designers, as they assumed things that (sometimes) don't hold. If on the other hand, the malfunction is the result of a failed assumption-needed, the problem rests with the users as they used the product outside its intended domain. As software is usually also an instance of such kind of products, provided by software engineers, and used by the software users, this distinction can also be used to illuminate disputes between software users and software engineers. For instance, with a different terminology, Twerski and his colleagues stress the key difference between design defect and failure-to-warn situations [18].

For a concrete example, on 4th June 1996, a flight of the Ariane 5 rocket ended in a crash, caused by the value of a particular variable exceeding its assumed limit. As reported in a post mortem study[19], the engineers underestimated possible environmental conditions, and '… was not analyzed or fully understood which values this particular variable might assume'.

In contrast, in the case that an assumption is interpreted as an assumption-needed, the assumption is adopted as a design component that describes the context in which the design solution works. This interpretation choice grants the assumptions-needed the ability to delimit the scope of the solution. This possibility is of substantial practical usefulness for the software engineers facing time and resource limitations. For example, an *intended user assumption* falls exactly into this sense of interpretation: there is a group of target users assumed by the engineers, and it is only for that target group that the software is expected to work [20]. From a legal point of view, as long as the proper disclaimers to the target users are made in a clear manner, the engineers of the software are covered in their legal responsibilities. That is to say that they are not liable for the effects of the software to the users outside the assumed target group.

According to the legal issues above, we can derive the importance of making clear these two types of assumptions in a software design. Without such a distinction, we don't know how to assign responsibility when some undesired consequence is brought about by the system's operations. In summary, a mature software engineering process should make explicit assumptions that underlie a design, just like requirements and functionalities are made explicit by specifications. In that respect, we argue that assumptions support *specifications* in satisfying *requirements*.

Capturing Assumptions Explicitly

Lewis stated that 'assumptions are made concerning how the software will be used, …, what environment it will operate in', as well as 'the incompatibilities between the assumptions and the assumed operation environment will cause failures' [2].

Based on this observation, Lewis proposes an *Assumption Management System* (AMS) to manage assumptions extracted from source code. This system is supported by the functions of storing the extracted assumptions in a repository, querying the repository, and making managerial decisions based on the assumptions recorded in the repository.

```
/*-
    <assumption>
        <type>
            Assumption type.
        </type>
        <description>
            Assumption description.
        </description>
    </assumption>
*/
```

Fig. 5: Syntax structure in Lewis' assumption management system

The syntax structure of an assumption assertion is adopted here from Lewis' work as shown in Fig. 5. Simply speaking, what Lewis did is to record assumptions as comments in source code, and the assumptions are encoded in a XML-style hierarchy. As shown in Fig. 5, and following the usual XML convention, the pair of '/* ... */' indicates the comment area, the pair of labels '<*assumption*>' indicates the assumption area, the pair of labels '<*type*>' indicates the type of the assumption, and the pair of labels '<*description*>' indicates the natural language description of the assumption.

The idea is that assumptions are captured and recorded by software engineers while they are writing source code. When the source code is ready for operation, a XML parser can be used to extract its assumptions and store them into a repository in a structured way for future queries. This is useful for sharing assumptions with all members in a software project, thereby reducing chances of misunderstanding, and ensuring consistency of the whole system.

Although, as stated by Lewis, recording and parsing of assumptions as contents of comments in source code has been

proved very useful in the coding process in a software engineering project, this is too late a stage for uncovering possible inconsistencies in the project. This is summarized by Lewis herself that 'to address interoperability requirements, the use of assumptions management would have to be moved to other activities and artifacts of software development, such as requirements analysis, architecture, and design'.

The problem pointed out by Lewis is essential to software engineering projects, as evidence suggests that errors in requirements, such as misinterpreting or neglecting some implicit assumptions, are much more expensive to fix at a later stage than in the early stages of a software project [21]. What we advocate here is that assumptions be captured and analyzed in the requirements engineering stage of the process, rather than later on, downstream.

Instead of dealing with source code, we capture and record assumptions in the process of deriving the requirement specification. In this stage, requirements are decomposed or refined into specifications including an external specification and an internal specification. Moreover, we provide here a more refined categorization of assumptions that should be discovered by software engineers in the requirements engineering stage. Furthermore, and more importantly, we illustrate the key role that such assumptions play in linking world and machine phenomena. To illustrate these points, in the following section, we present an example of a meeting

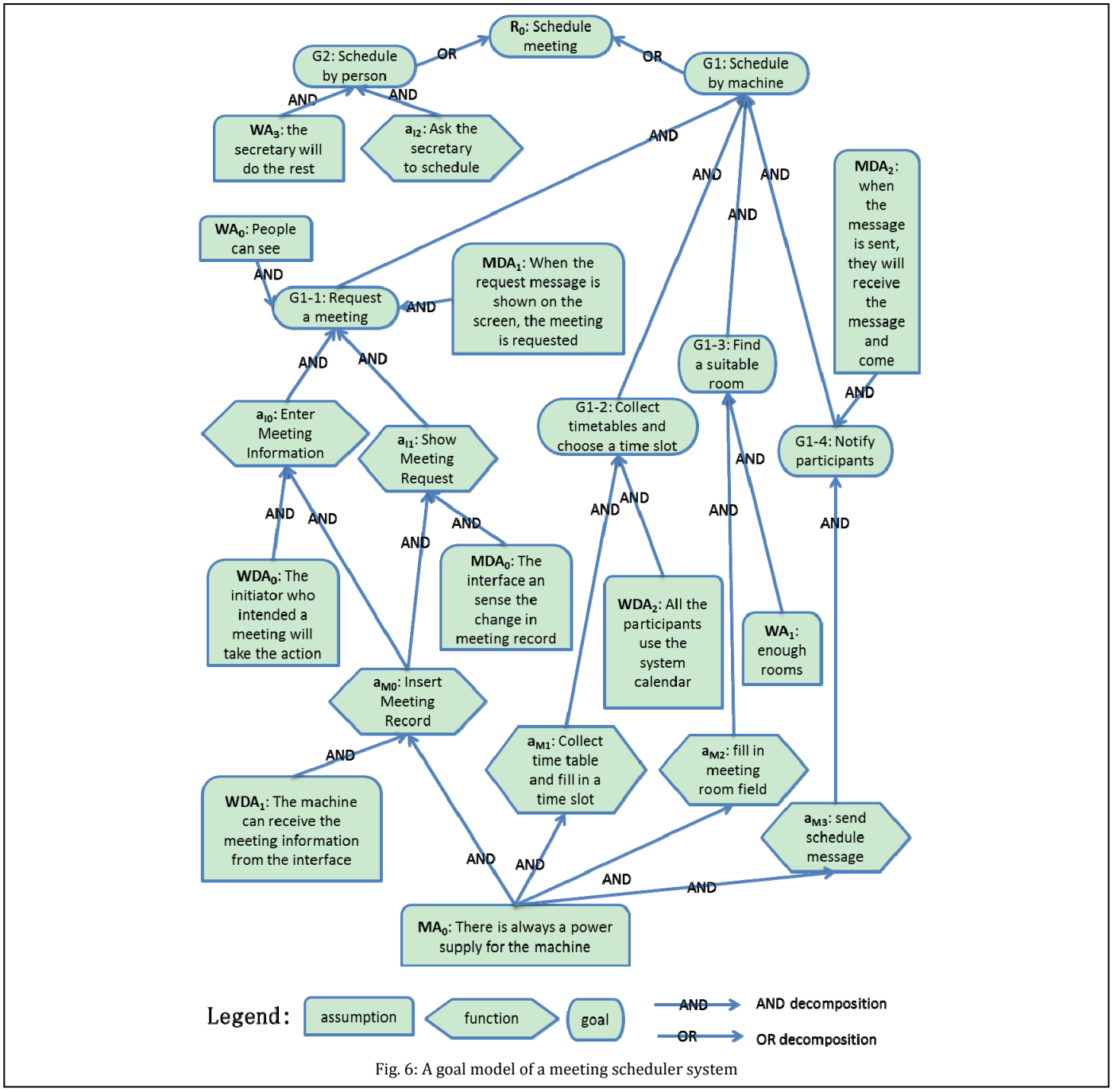

Fig. 6: A goal model of a meeting scheduler system

scheduler system that demonstrates how software engineers can capture and represent these different kinds of assumptions in a requirements engineering process.

V. THE MEETING SCHEDULER EXAMPLE

As mentioned in the preceding section, instead of dealing with assumptions in the source code, we emphasize the importance of capturing assumptions in the requirements engineering stage, to detect errors as early as possible.

Specifically, the requirements engineering process can be further divided into two phases as stated by Yu [22], namely, an early phase and a late one. In the early phase, software engineers collect requirements from stakeholders. Usually, in this phase, social activities (e.g., such as interviews, surveys, and etc.) are used to gather informal and vague requirements, trying to determine whether a software system should be developed. The output of this phase is typically a document in natural language that summarizes relevant information.

In the late phase, the initial requirements gathered in the early-phase are used as input, and further analysis is carried out to check their feasibility and consistency. A goal model [22] can be adopted in this phase, where requirements are represented as goals that can be decomposed into sub-goals and then into tasks. These tasks are operational at the interface between the world and the machine, and we regard these tasks as the interface actions belonging to an external specification.

Usually, the decomposition process of a goal model stops here, and one of the main reasons is that the researchers and engineers want to avoid the so-called '*Implementation Bias*' (as discussed, for instance, by Jackson et al. [7]). The general idea is that implementation details have no place during requirements engineering.

We agree that it is important to make clear the boundary between world and machine. However, it is also essential to indicate explicitly the link across the boundary, or we will fall into the world-on-machine paradox. As software can only directly manipulate the virtual variables in a machine, the sense in which the result of this manipulation affects the outside world is neglected by the view of avoiding implementation bias.

As discussed, to address the paradox we need to explicitly capture links between world and machine. As the reader can see in Fig. 6, a simple meeting scheduler system is adopted as our example, and its requirements are represented by a goal model. Being different from the literature work [22], in the notation used in this model, we capture not only requirements and interface actions that are part of the external specification, but also the internal specification with machine actions. Moreover, the four different kinds of assumptions are explicitly represented in the model in order to link the world and the machine together.

We choose the meeting scheduler system as our example in this paper, because it is a well-known exemplar in the Requirements Engineering literature [22], [23], [24]; and on the other hand, it is a relatively simple scenario, that most readers are familiar with.

Requirements (R): As mentioned in section 3, our single requirement is that 'a meeting shall be scheduled once a meeting initiator intends to schedule one'. This requirement is labeled as R_0 in the goal model. The situation calculus formalization of R_0 has already been shown in section 3, hence, we do not repeat it here. We treat the other expressions in the same way, only the newly proposed ones will be encoded into situation calculus in this section.

The requirement R_0 is refined into two alternative sub-goals through 'OR decomposition' links: 1) the meeting initiator can do it manually, asking a secretary to arrange a meeting as indicated by the interface action a_{I2}. This choice of solution needs a world assumption WA_3 to ensure the reliability of the secretary who should arrange everything else for the meeting initiator; 2) however, the secretary might be overloaded, such that she/he cannot be relied to do this work quickly and correctly. If this is the case, the second choice of adopting a meeting scheduler system might be preferable.

The scenario of using this system is briefly summarized as follows: when a meeting initiator wants to schedule a meeting, she will make a request to the system. For every request, the meeting initiator enters the meeting information into the system through the interface, including a list of intended participants, the title of the meeting, and etc. To schedule the meeting, the system needs to collect timetables from all participants, choose a time slot, and then assign a meeting room for the meeting. Finally, the system also must inform all the participants in the provided list.

External Specification (S_{ext}): according to the scenario aforementioned, the sub-goal '*G1*: Schedule by machine' could be refined further into four sub-goals, including '*G1-1*: Request a meeting', '*G1-2*: Collect timetables and choose a time slot', '*G1-3*: Find a suitable room', '*G1-4*: Notify participants'. To simplify the work of the initiator as much as possible, it is ideal to design such a solution that the only work the initiator would need to do is to make a meeting request to the system with the necessary meeting information. By receiving that request, the system would do everything else for the initiator.

In this case, it seems that the newly proposed system is a complete replacement of the secretary for scheduling meetings. The system provides two interface actions a_{I0} and a_{I1} to implement the sub-goal *G1-1*. Through these actions, the initiator can submit a meeting request to the system and get a request confirmation message from the system. To ensure the fulfillment of this sub-goal through executing these two interface actions, four assumptions are included in the model: WA_0, WDA_0, MDA_0 and MDA_1.

All these elements mentioned here (a_{I0}, a_{I1}, WA_0, WDA_0, MDA_0 and MDA_1) have already been translated into situation calculus expressions in the example that explains the chaining mechanism formed by such actions and assumptions as shown in Fig. 4. Here we only show how they could be asserted in a goal model, and the corresponding translations to situation calculus expressions are not repeated. As another contribution, in the sequel, we explain further how the sub-goals (e.g., *G1-2*, *G1-3*, and *G1-4*) in the goal model could be decomposed into machine actions (e.g., a_{M1}, a_{M2}, and a_{M3}) with the help of different kinds of assumptions.

Internal Specification (S_{int}): the internal specification of the meeting scheduler system contains several machine actions, including a_{M0}, a_{M1}, a_{M2}, and a_{M3}. The first one has already been explained in section 4, and, as previously mentioned, we here analyze the last three of them.

$$a_{M1} : Collect\_Timetables\_and\_Fill\_Time\_Slot(meeting)$$
$$PRE : Poss(a_{M1}, s) \leftrightarrow$$
$$meeting\_record\_inserted_M(meeting, s)$$
$$POS : Poss(a_{M1}, s) \rightarrow$$
$$[meeting\_time\_slot\_filled_M(meeting, s')$$
$$\wedge s' = do(a_{M1}, s)]$$
$$WDA_2 : use\_system\_calendar_W(participant)$$

Firstly, the sub-goal *G1-2* is implemented by the machine action a_{M1}. Whenever there is a meeting record inserted into the database, this action a_{M1} will be executed as indicated by its precondition. By executing this action, all the participants' timetables will be automatically collected, and as a calculating result, a suitable time slot will be filled into the meeting record (as indicated in this action's post-condition).

However, this action only works in the situations where all the participants use the system calendars, so we introduce an assumption WDA_2 to constrain the operational situations of this system. This assumption presupposes some interactions between the participants and the machine. In particular, it assumes that the operation of the machine depends on some world phenomena, hence we treat it as a world dependence assumption as indicated in the corresponding situation calculus expression.

$$a_{M2} : Assign\_Meeting\_Room(meeting)$$
$$PRE : Poss(a_{M2}, s) \leftrightarrow$$
$$meeting\_time\_slot\_filled_M(meeting, s)$$
$$POS : Poss(a_{M2}, s) \rightarrow$$
$$[meeting\_room\_filled_M(meeting, s')$$
$$\wedge s' = do(a_{M2}, s)]$$
$$WA_1 : enough\_room_W(meeting)$$

Then, the sub-goal *G1-3* is implemented by the machine action a_{M2}. The post-condition of a_{M1} is used as the pre-condition of a_{M2} such that whenever the time slot of a meeting record is provided, the action a_{M2} is executed. By executing a_{M2}, a new situation will be reached, in which a room number is assigned to the meeting record as shown in its post-condition.

As mentioned several times in this paper, to simplify the work of engineers in this modeling case, the system only deals with the situations in which there are enough rooms, and it will not work properly in a context in which this is not the case. Thus, a world assumption WA_1 is introduced such that for every meeting that needs to be scheduled, it is assumed that there are always rooms available.

$$a_{M3} : Notify\_Particiapnts(meeting)$$
$$PRE : Poss(a_{M3}, s) \leftrightarrow$$
$$meeting\_record\_inserted_M(meeting, s)$$
$$\wedge meeting\_time\_slot\_filled_M(meeting, s)$$
$$\wedge meeting\_room\_filled_M(meeting, s)$$
$$POS : Poss(a_{M3}, s) \rightarrow$$
$$[notification\_messages\_sent_M(meeting, s')$$
$$\wedge s' = do(a_{M3}, s)]$$
$$MDA_2 : notification\_messages\_sent_M(meeting, s)$$
$$\leftrightarrow meeting\_scheduled_W(meeting, s)$$

Finally, the sub-goal *G1-4* is implemented by the machine action a_{M3}. Whenever the meeting record is inserted, filled with a time slot and a room number, the action a_{M3} is executed as indicated by its pre-condition. By executing it, the meeting schedule notification message is sent to all the participants as shown in its post-condition.

However, sending messages is not the same as confirming the meeting with the participants, i.e., the interface action by itself is not directly equivalent to the social action of creating a meeting schedule (a social object) involving all those participants. Once more this gap is filled by a machine dependence assumption MDA_2, which links the *message sending* and the *schedule confirming*. In other words, according to this assumption, the solution is simplified, and whenever the messages are sent, we assume the participants will receive them, confirm them, and at the same time the meeting is also scheduled. Or put it in yet different terms, the message sending *counts as* a schedule being confirmed in this context.

In summary, through the meeting scheduler example, we have demonstrated how the analysis of assumptions can be introduced in a requirement engineering process instead of only in the code writing process. Additionally, four different kinds of assumptions are represented in a goal model, linking the world and the machine together. In this model, we also show how these assumptions can be formally represented.

VI. Related Work

In [15], Lamsweerde also recognizes Jackson's work, and treats both the assumptions-used (hypothesis) and assumptions-needed (expectations) as two kinds of assumptions that should be addressed in a software engineering process. In that paper, even a similar extension of Jackson and Zave's original formula is provided: *{Specification, Assumption, Domain Property}* $\models$*Requirement.* However, in that paper, the author focuses only on analyzing world assumptions and machine assumptions. Here, instead, we propose two additional kinds of dependence assumptions which, as we have shown, are essential for explaining how software can affect the social world.

Another work that systematically analyzed the role of assumptions in requirements engineering is the work of Jureta reported in [25]. The author, however, neither distinguishes

between assumptions-used and the assumptions-needed, nor provides a classification of assumptions into different kinds as we do in this paper. His work instead focuses on matching requirements engineering related concepts to different kinds of mental states. For instance: an assumption is matched to a believed proposition; a requirement is matched to a desired proposition; and a task is matched to an intended proposition. As we previously mentioned, we intend to address such a detailed ontological analysis of the nature of assumptions as well as of their relations to cognitive and social agents in a companion paper.

As stated in section 4, Lewis [2] proposes an Assumption Management System to help practitioners to record, query, and validate assumptions. However her work only deals with assumptions in source code (implementation phase). In contrast, we here advocate that assumptions should be elicited, represented, analyzed in the requirements engineering phase of this process. This allows software engineers to catch errors involving mistaken assumptions as early as possible in the process. Moreover, differently from Lewis' work, we propose here a finer-grained classification of assumptions and elaborate on their key role in linking the world and machine states.

VII. Conclusion and Future Work

The contributions of this paper are three-fold: 1) we propose a preliminary ontology of assumptions including four kinds of assumptions. Based on this classification, we elaborate on the role of these assumptions in explaining how social facts can be affected by the manipulation of symbolic structures in a machine; 2) we clarify the concept of 'assumption' adopted in software engineering literature, and emphasize the importance of clarifying the interpretation of the assumptions as either *assumptions-used*, or as *assumptions-needed*; 3) as a methodological contribution and, by employing a meeting scheduler example, we demonstrate how assumptions can be explicitly and systematically elicited and represented as a part of the requirements engineering process. As this example demonstrates, requirements engineering, in particular, and software engineering in general can benefit from an awareness of the four kinds of assumptions.

Assumptions are of fundamental importance to software engineering. Therefore, we advocate further research in order to develop a clearer understanding of what assumptions are. Accordingly, on the theoretical side, we intend to publish a dedicated paper exploring the ontological nature of assumptions as well as systematizing the nature of the relations to other elements of our software ontology [8], [9]. On the practical side, the work proposed here opens up the possibility of developing a next generation of *assumption management system*s that covers important software artifacts, including requirements, architectures and more. In these systems, the management of assumptions could be integrated with the requirements engineering process. As a consequence, errors involving assumptions could be identified and addressed earlier and more effectively. Moreover, these systems could also support decision-making activities by software stakeholders, from owners to developers, to end users.

Acknowledgments

Support for this work was provided by the ERC advanced grant 267856 for the project entitled "Lucretius: Foundations for Software Evolution" (http://www.lucretius.eu), as well as the "Science Without Borders" project on "Ontological Foundations of Service Systems" funded by the Brazilian government.

References

[1] M. Jackson and P. Zave, "Deriving specifications from requirements: an example," in *Proceedings of the 17th international conference on Software engineering*, 1995, pp. 15–24.

[2] G. Lewis, T. Mahatham, and L. Wrage, "Assumptions Management in Software Development," Pittsburgh, PA, 2004.

[3] M. A. Al Mamun and J. Hansson, "Review and Challenges of Assumptions in Software Development," in *Second Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS)*, 2011.

[4] D. Brown, "Assumptions in Design and Design Rationale," in *DCC'06 Workshop on Design Rationale: Problems and Progress*, 2006.

[5] T. Tun, R. Lutz, B. Nakayama, Y. Yu, D. Mathur, and B. Nuseibeh, "The role of environmental assumptions in failures of DNA nano systems," in *Proceedings of Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS 2015) co-located with ICSE 2015*, 2015.

[6] J. McCarthy and S. U. C. S. D. A. I. Laboratory, *Situations, actions, and causal laws*. Comtex Scientific, 1963.

[7] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 1, pp. 1–30, 1997.

[8] X. Wang, N. Guarino, G. Guizzardi, and J. Mylopoulos, "Towards an Ontology of Software: a Requirements Engineering Perspective," in *8th International Conference on Formal Ontology in Information Systems*, 2014, pp. 317–329.

[9] X. Wang, N. Guarino, G. Guizzardi, and J. Mylopoulos, "Software as a Social Artifact: A Management and Evolution Perspective," in *33rd International Conference on Conceptual Modeling*, 2014, vol. 8824, pp. 321–334.

[10] J. Ryan-Collins and T. Greenham, "Where does money come from."

[11] D. Franken, A. Karakus, and J. G. M. Michel, *John R. Searle: Thinking About the Real World*. De Gruyter, 2010.

[12] M. Jackson, "The World and the Machine," in *Proceedings of the 17th International Conference on Software Engineering*, 1995, pp. 283–292.

[13] C. A. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *Software, IEEE*, vol. 17, pp. 37–43, 2000.

[14] M. Jackson, "Specialising in Software Engineering," in *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, 2007, pp. 3–10.

[15] A. Van Lamsweerde, "From Worlds to Machines," in *A Tribute to Michael Jackson*, Lulu Press, 2009.

[16] R. Reiter, "Logic and Data Bases," H. Gallaire and J. Minker, Eds.

Boston, MA: Springer US, 1978, pp. 55–76.

[17] R. Ennis, “Identifying implicit assumptions,” *Synthese*, vol. 51, no. 1, pp. 61–86, 1982.

[18] A. D. Twerski, A. S. Weinstein, W. A. Donaher, and H. R. Piehler, “Use and Abuse of Warnings in Products Liability-Design Defect Litigation Comes of Age,” *Cornell L. Rev.*, vol. 61, p. 495, 1975.

[19] J.-L. Lions, “Flight 501 failure,” *Rep. by Inq. Board*, 1996.

[20] M. Schultz, W. Conshohocken, C. Hill, L. Vegas, L. Angeles, S. Diego, and S. Francisco, “Defenses in a Product Liability Claim,” 2002.

[21] B. Nuseibeh and S. Easterbrook, “Requirements engineering: a roadmap,” in *Proceedings of the Conference on The Future of Software Engineering*, 2000, pp. 35–46.

[22] 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*, 1997, p. 226–.

[23] V. E. Silva Souza, “Requirements-based Software System Adaptation,” University of Trento, 2012.

[24] A. van Lamsweerde, R. Darimont, and P. Massonet, “Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt,” in *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, 1995, pp. 194–203.

[25] I. J. Jureta, J. Mylopoulos, and S. Faulkner, “A core ontology for requirements,” *Appl. Ontol.*, vol. 4, no. 3, pp. 169–244, Jan. 2009.