Noname manuscript No.
(will be inserted by the editor)

CaRE: A Refinement Calculus for Requirements Engineering based on Argumentation Theory

Yehia ElRakaiby · Alexander Borgida · Alessio Ferrari · John Mylopoulos

Received: date / Accepted: date

Abstract The Requirements Engineering (RE) process starts with initial requirements elicited from stakeholders—however conflicting, unattainable, incomplete and ambiguous—and successively refines them until a consistent, complete, valid, and unambiguous *specification* is reached. This is achieved by balancing stakeholders' viewpoints and preferences to reach compromises through negotiation. Several frameworks have been developed to support this process in a structured way, such as KAOS, i*, and RationalGLR. However, none provides the means to model the dialectic negotiation inherent to the RE process, so that the derivation of specifications from requirements is fully explicit and traceable. To address this gap, we propose CaRE, a refinement *calculus* for requirements engineering based on argumentation theory. CaRE casts the RE refinement problem as an iterative argument between all relevant stakeholders, who point out defects (ambiguity, incompleteness, etc.) of existing requirements, and then propose suitable refinements to address them, thereby leading to the construction of a refinement graph. This graph is then a conceptual model of the RE process. The semantics of refinement graphs is provided using Argumentation Theory, enabling reasoning over the RE process and the automatic computation of software specifications. An alternate semantics is also presented based on abduction and using Horn Theory. The application of CaRE is showcased with an extensive

Y. ElRakaiby
Université du Luxembourg, Luxembourg
E-mail: yehia.elrakaiby@gmail.com

A. Borgida
Rutgers University, New Brunswick NJ, USA
E-mail: borgida@rutgers.edu

A. Ferrari
CNR-ISTI, Pisa, Italy
E-mail: alessio.ferrari@isti.cnr.it

J. Mylopoulos
University of Toronto, Canada and University of Trento, Italy
E-mail: jm@cs.toronto.edu

example from the railway domain, and a prototype tool for identifying specifications in a refinement graph is presented.

Keywords requirements engineering · requirements refinement · RE process · RE calculus · argumentation theory · formal semantics

1 Introduction

Software requirements engineering (RE) constitutes a critical phase for any software development project. The original core problem in RE consists of transforming the initial requirements R elicited from stakeholders—however informal, ambiguous, unattainable, etc.— through a systematic refinement process into a specification S that (a) consists of functional requirements, quality constraints and domain assumptions, (b) is consistent, complete, and realizable, and (c) fulfills or satisfies R.

Variants of this problem constitute the backbone of RE research, and several conceptual notations and techniques have been proposed since the 70s (e.g., Structured Analysis by Douglas Ross [41]) to engineer the refinement process. To begin with, consider two RE techniques that can be viewed as research baseline and analogues for our work, namely SADT [41] and the family of goal-oriented RE (GORE) techniques [9,49]. In each case, we note 1) the basic ontology underlying each approach, 2) the refinement process by which the requirements are built, and 3) the "requirements document" resulting from the enactment of this process.

SADT (1977) [41] was the first widely known requirements specification notation and methodology. The modeling ontology of SADT consists of *data* and *activity boxes*, connected by *input/output/control arrows*. The refinement methodology is a structured decomposition of non-atomic boxes into labelled sub-boxes, which are interconnected by labelled arrows in appropriate ways. Therefore, the final requirements document/model consists of a number of pages, each describing the internal content of a box; all unexpanded boxes are viewed as atomic/realizable. Ross [41] explicitly stated that SADT can be used to describe not just software requirements but to communicate any idea/conceptual model, and showed how to describe the process of model building in SADT itself. **RML (1992)** [22] expanded the ontology of SADT by adding *assertion objects*, and extended its methodology with refinement by specialization, supported by subclass hierarchies and inheritance. Rather than using boxes, RML's language was explicitly based on Knowledge Representation, and formalized in logic.

Goal-Oriented RE—GORE (1993) [9,49] is an influential RE paradigm which, in its simplest form, has an ontology consisting of *goals*, connected by *reduction* and *conflict relations*. The methodology suggests refining non-realizable goals (ones that cannot be implemented directly) using AND/OR decomposition. The final requirements model then consists of a graph of goals and decomposition relationships, with operationalizable goals as leafs.

All prior RE approaches, starting with [30], viewed initial requirements R as being *satisfied* by specification S under domain assumptions A, if A and S together logically entailed R. This notion of fulfillment runs counter to requirements engineering practice, where stakeholder requirements are routinely weakened because they

are unnecessarily strong (e.g., "system shall be available 7/24"), or even dropped altogether. Such decisions are not accounted for in existing requirements engineering models and frameworks, often resulting in an incomplete documentation of the process, lack of transparency and traceability with respect to important decisions or design choices made, and a difficulty to trace and analyze the evolution of the refinement process at later-stages of software development. Indeed, incomplete and hidden requirements have been recognized as one of the most common and challenging problems in RE practice [17].

To address the aforementioned limitations of existing methods, the present paper proposes an approach, called **CaRE**, whose ontology consists of *requirements/goals*, *defects* and *refinements*, the latter two of various sub-types. CaRE offers a novel calculus of operators that can be used to critique requirements using various defect types, and to address such defects using assorted refinements[1] The CaRE refinement methodology suggests viewing the use of the operators as a dialectic argument between stakeholders, including requirements engineers, each of whom may point out defects in the current requirements, or add new requirements that address posited defects. The result of enacting this argument is a *refinement graph*, which records requirements, defects and refinements, and for which we define a notion of "acceptability" that replaces the notion of "satisfaction"/"fulfillment" for GORE approaches.

The set of defect subtypes in CaRE is inspired by the IEEE/ISO Standards on Software Requirements Specifications (SRS) [1,2], which have detailed "defects" to be addressed during the software requirements refinement process. The set of refinements addressing them is gathered from the RE literature, which contains many proposals for dealing with *specific* types of defects. These include techniques for eliminating forms of conflict, such as inconsistencies [28] and obstacles [45]. Still others focus on recognizing ambiguity introduced by natural language, for example [43]. Such refinements can't be accounted for explicitly by proposals in the literature. Note that basic GORE technique is covered in CaRE by the **nonAtomic** defect (marking non-operationalizable goals) and the **reduce** operator, which can be used to perform AND-decomposition of a goal.

The CaRE process results in a refinement graph with a set of nodes R, each representing a requirement; some are initial, some are leaf nodes, with no defects, and others are intermediate requirements. A specification set S, consists of some leaf nodes, which is said to address R if there is an *"acceptable argument"* that involves refining S from R. This renders the derivation of S from R a Hegelian dialectic process of thesis-antithesis-synthesis [25], also similar in spirit to the inquiry cycle [40], though our proposal includes more structure, technical detail, and reasoning support. Addressing a given set of requirements by offering an acceptable argument is a weaker notion of fulfillment than satisfying it, because it allows a requirement to be weakened or altogether dropped, as long as there is an acceptable argument for this. Towards this end, we adopt argumentation semantics from Dung [11], and we provide an alternate semantics through abduction, based on a Horn Theory involving requirements and handled defects [42].

[1] It is important to note that the term "refinement" means "improvement", not just "elaboration", so that requirements can be weakened or even dropped.

Furthermore, prior RE approaches do not account for defects—the rationales behind refinements. In contrast, in CaRE, defects are explicit first-class entities, providing a comprehensive documentation of the RE refinement process and improving its review and comprehension by stakeholders. In addition, building of refinement graphs in CaRE is a *monotonic process*, not requiring modification nor revision of previous steps, as identification of defects allows a disciplined refinement or even rejection of previously introduced refinements. This greatly simplifies *change management*, a major challenge in requirements engineering which most current frameworks fail to address adequately.

The calculus is intended to be used in a process that involve all stakeholders (or a representative subset thereof in case of large numbers) who consider the initial stakeholder requirements and iteratively critique and refine them to generate new requirements that remove defects. The process terminates when there is a derived specification that is acceptable to all concerned.

The CaRE proposal fits well the paradigm of model-driven software engineering and tackles the first step of the software engineering process: going from stakeholder requirements to a specification. This step is different from others that follow downstream in that stakeholder participation is essential and there are no prospects for full automation. All that can be hoped for, which CaRE supports, is suitable conceptualization and systematization of the process, along with reasoning support over models of process enactment.

The contributions of this work include:

- A comprehensive refinement calculus for RE, inspired by goal-oriented RE but which adds: (i) "defects" and "refinements" to its ontology, based on a full set of defect types from IEEE/ISO standards; (ii) a comprehensive set of refinement operators for defects; (iii) refinement graphs, which are conceptual models of the RE process enactment, and can serve as explanation/rationale for derived specifications.
- An argumentation-based semantics of what it means for a specification to address a set of stakeholder requirements, and an equivalent semantics based on Horn abduction. The systematic process for constructing CaRE refinement graphs, inspired by its argumentation-based semantics, supports negotiation and convergence towards agreement among stakeholders. This leads to an environment where all stakeholders are involved, in contrast with most previous approaches, where only requirements engineers conduct requirements analysis.
- Reasoning support that, given an initial set of requirements R and a constructed refinement graph, returns all specifications S that address R. This is implemented as a prototype tool that is available on the web.
- A scenario from the railway domain illustrating the elements of our calculus and how they can be used to derive specifications from requirements.

This paper extends a recent contribution to the ER conference [13]. With respect to the original submission the current paper adds more information and explanation about the proposed calculus, a complementary abduction-based semantics (Sect. 4) which is provably equivalent in simpler cases, a realistic application scenario based

on the railway domain in which CaRE is showcased (Sect. 5), and an extended related work section to better frame our work with respect to existing literature (Sect. 6).

The remainder of the paper is structured as follows. In Sect. 2, we illustrate the fundamental ingredients of the proposed calculus, namely defects, operators, and refinement graphs. In Sect. 3, we provide a semantics for the calculus based on Argumentation Theory. Sect. 4 presents the alternative semantics based on abduction. Sect. 5 showcases the calculus on a realistic scenario. Sect. 6 presents related works, while Sect. 7 concludes and discusses limitations and future work.

2 CaRE Requirements Calculus

The proposed approach consists of a calculus and a systematic process for requirements critiquing and refinement. The calculus is based on a collection of defect types and refinements. The defect types are inspired by the IEEE/ISO standards and represent issues that could be identified by stakeholders for one or more requirements. Refinements, on the other hand, are the means for fixing defects. By means of an iterative process of defect identification and refinement, a refinement graph is constructed and zero or more specifications are derived.

2.1 CaRE Metamodel

Figure 1 shows a metamodel of CaRE refinement graphs, composed of: (1) ***Requirement***s, (2) ***Defect***s, and (3) ***Refinement***s. We distinguish initial requirements gathered at the start of the refinement process, and those with no defects. In the metamodel, this is represented using the Boolean attributes *isInitial* and *isFinal* respectively. The various elements of the metamodel are detailed in the following.

Defect Types: Defects are classified into single target defects that critique a single requirement, (e.g. "System shall be user-friendly" is ambiguous), and multi-target defects that critique a collection of requirements (e.g., {"System shall be user-friendly", "System shall be highly secure"} are conflicting). Moreover, the defect hierarchy in the CaRE metamodel is disjoint and complete in the sense that each defect belongs to exactly one defect type.

The single target defects are:

- **nonAtomic**: target requirement is not operationalisable. For example, $\langle g_1$:"System shall schedule a meeting upon request"$\rangle$ may not be atomic since there is no single action the system-to-be or an agent in the environment can perform to address it.
- **ambiguous**: the requirement admits many interpretations because it is vague, imprecise, or otherwise ambiguous. For example, $\langle g_2$:"The authentication process shall be easy"$\rangle$ is ambiguous since the term *easy* is vague.
- **unattainable**: the requirement is not feasible, i.e. doesn't have a realistic solution. For example, $\langle g_3$:"The system shall be available at all times"$\rangle$ is unattainable because it assumes eternal availability of power and other resources.

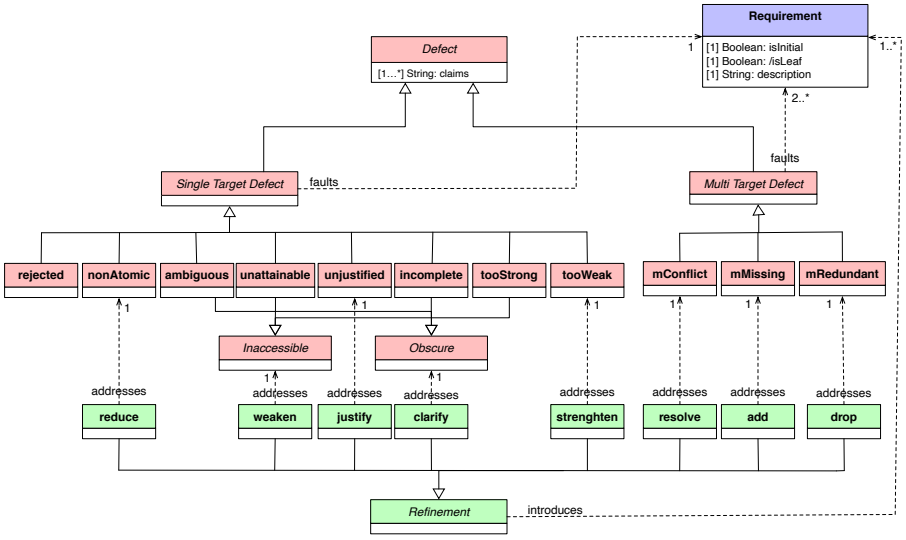

Fig. 1 CaRE Metamodel

– **unjustified**: the requirement does not have an explicit motivation. For example, $\langle g_4$:"The system shall run on Windows operating system"$\rangle$ is missing an explicit justification why other operating systems are not considered.
– **incomplete**: the requirement is missing information. For example, $\langle g_5$:"In case of fault, the system shall send an error message"$\rangle$ is incomplete because it does not specify a recipient for the message.
– **tooStrong**: the requirement is over-restrictive. For example, $\langle g_6$:"The website shall use HTTPS protocol"$\rangle$, may be too strong if there is no sensitive data.
– **tooWeak**: the requirement is too weak. For example, $\langle g_7$:"The DB system shall process 500 transactions/sec"$\rangle$ is too weak if the expected workload for the system-to-be is 1,000 transactions/sec.
– **rejected**: the requirement is unacceptable and is therefore rejected. For example, in the context of an app recommending nearby restaurants to users, a requirement such as $\langle g_8$:"The app shall support chatting between user and restaurant"$\rangle$ may be deemed unacceptable to a restaurant stakeholder.

The multitarget defects are:

– **mConflict**: the target set of requirements doesn't admit any solutions, even though subsets may do so. For example, the requirements $\langle g_9$:"The train control system shall stop the train if a red signal is missed"$\rangle$ and $\langle g_{10}$:"The train control system shall not apply brakes if the speed is below 30 km/h"$\rangle$ are conflicting if a red signal is missed while the train's speed was less than 30 km/h.
– **mMissing**: the set of requirements is incomplete. For example, a set of requirements for a social network platform is mMissing if it does not include any privacy requirements.

– **mRedundant**: here a set of requirements is too strong or redundant, as in $\langle g_{11}$:"The system shall support authentication through fingerprint recognition"$\rangle$ and $\langle g_{12}$:"The system shall support authentication through iris recognition"$\rangle$.

IEEE Software Requirement Specification The set of defects in CaRE is inspired by the IEEE Standard 830-1998 on Software Requirements Specifications (SRSs) which lists 8 desirable properties are proposed for SRSs: *Correctness*, *Unambiguity*, *Completeness*, *Consistency*, *Ranking*. *Verifiability*, *Modifiability* and *Traceability*. Of these, the first four are addressed by CaRE at a finer granularity to make CaRE more user-friendly. Specifically, Correctness is decoupled into **tooStrong**, **tooWeak**, **mRedundant**, Unambiguity by **ambiguous**, Completeness by **unjustified**,**incomplete** and **mMissing** while Consistency is decoupled into **unattainability** and **mConflict**. Three other desired properties of the standard (*Ranking*, *Modifiability*, *Traceability*) concern the SRS document rather than requirements. Finally, *Verifiability*, was deemed to be subsumed by *Unambiguity*. The CaRE list of defects also includes non-atomicity which is the defect addressed by GORE.

Refinement Operators: A refinement operator invocation, $op(d, newR)$, addresses a defect d of some existing requirement(s), and introduces new requirement(s) $newR$ that address the defect and are hopefully acceptable. Note that currently $newR$ may contain an existing requirement as long as this doesn't introduce cycles in the refinement graph.

Each refinement operator concerns a (set of) defective requirement(s), and is applicable to one or more defect types. Conversely, each defect type has at least one refinement operator that is applicable to it, i.e., can eliminate defects of that type. Defects of type **rejected** are an exception: in this case there is no possible fix, as the rejected requirement constitutes a dead end. Note that the set of operators is not "minimal" – some operators behave similarly, but we have chosen to keep them in order to make the calculus more readily usable. The operators are as follows:

– **weaken**: introduces a weaker requirement. For example, the unattainable requirement g_3 may be weakened into $\langle g_{13}$:"The system shall be available at all times, with interruptions of $\leq$2 hours"$\rangle$. **weaken** is applicable to defects of the type *Inaccessible*, which concretely are **tooStrong** or **unattainable** defects.
– **strengthen**: introduces a stronger requirement. For instance, g_7 may be strengthened into $\langle g_{14}$:"The system shall process 1,200 tps"$\rangle$. **strengthen** is applicable to defects of type **tooWeak**.
– **reduce**: decomposes a requirement into a set $g_1, ..., g_n$ using AND-refinement. **reduce** is applicable to defects of type **nonAtomic**.
– **add**: is applicable to defects of type **mMissing**, and introduces a new requirement (that was missing), as well as copies of its input requirements.
– **clarify**: is applicable to *Obscure* defects, which, concretely, include **incomplete** and **ambiguous** defects; and introduces a, presumably, improved requirement.
– **justify**: introduces a new requirement that represents an explicit motivation for another requirement, and is applicable to **unjustified** defects.
– **resolve**: applies to defects of type **mConflict**, typically moderating or dropping some of the original requirements.

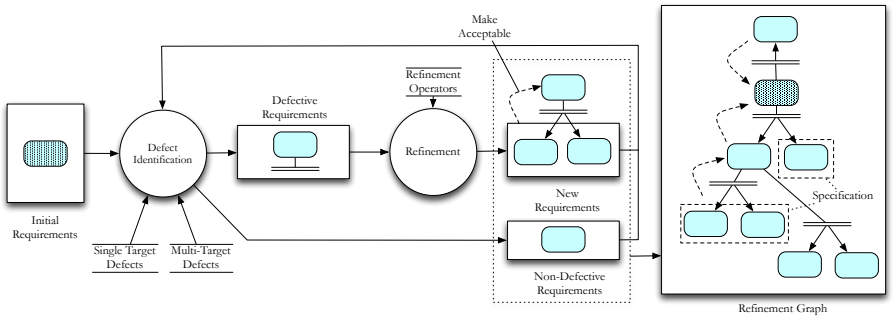

Fig. 2 Overview of the Approach

– **drop**: given a set of **mRedundant** requirements, produces copies of a proper subset of R excluding redundant elements.

2.2 Incremental Construction of a Refinement Graph

In this section, we provide an overview of the incremental process of building a refinement graph, and give intuitions of what it means for a requirement to be acceptable. Formal definitions appear in Sect. 3.

Fig. 2 depicts the steps that lead to the construction of a refinement graph. Given an initial set of requirements, these are critiqued by stakeholders to identify defects (step Defect Identification). Then, requirements – or sets of requirements – that have defects are refined by applying refinement operators (step Refinement), producing new requirements. If the new requirements are acceptable to all stakeholders, i.e. have no defects, their respective *isFinal* attribute is set to true. This process is repeated until no new defects are identified, or all stakeholders are satisfied with derived specifications. Thus, the result of the process is a *refinement graph* where some of the leaf nodes are requirements that have no defects, are therefore acceptable and are labelled as such. Intuitively, acceptability of requirements is propagated from leaf nodes towards higher-level nodes: whenever every defect of a requirement g has been addressed by a refinement with requirements which themselves are acceptable, g is acceptable. (A formal definition of "acceptable" is given in Section 3.) Finally, *specifications* of a refinement graph are determined by identifying minimal sets of leaf nodes[2] that make the initial requirements acceptable.

2.3 Graphical Notation and Running Example

This section presents a simple example used to illustrate our proposal and its graphical expression. The basic elements of the graphical notation are shown in Fig. 3, and consist of requirements, defects (single- and multi-target), and refinements. Each instance of these elements is associated with a unique id (REQ-id, DEF-id, REF-id in Fig. 3). As the elements are mostly self-explanatory, we will illustrate them through the example.

[2] Identification of leaf nodes is detailed in Sect. 3.

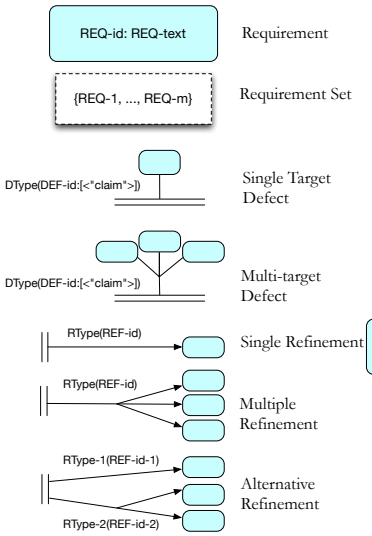

Fig. 3 Graphical Notation

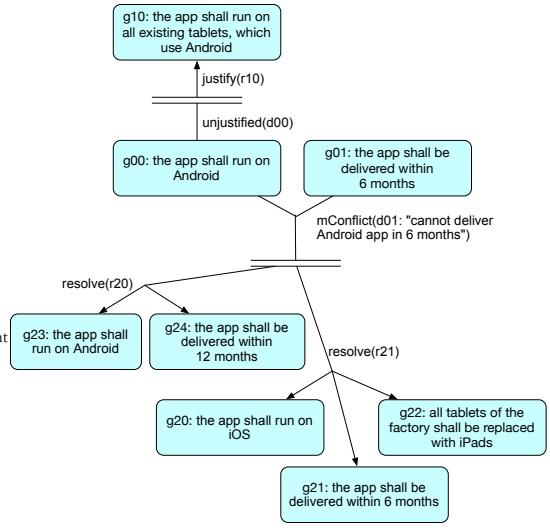

Fig. 4 Refinement Graph Example

The example, which is based on a simple elicitation case, is represented by the refinement graph in Fig. 4. In the running example, a customer requires a new app to be installed on the tablets of factory workers, to be used for sharing workflow information. The customer requires that the app runs on Android (g00). Furthermore, the customer wants the system to be delivered within six months (g01). The requirements analyst asks why Android is required (**unjustified**(d00) defect), and the customer replies that the tablets currently used by the workers are all Android tablets (g10, introduced with the **jusify** refinement r10). The requirements analyst knows that their software company has a very similar app for iOS, but that porting and adaptation would require twelve months. Hence, g00 and g01 are considered conflicting. In the refinement graph, an **mConflict** defect is specified, and a textual motivation (the optional <"claim"> in Fig. 3) is used to explain the nature of the defect: **mConflict**(d01:"cannot deliver Android app in 6 months"). To comply with the deadline, the requirements analyst suggests to develop the app for iOS (g20), so that its adaptation to the customer's needs is feasible within 6 months (g21). However, this would require replacing the tablets at the factory with iPad tablets (g22). Alternatively, the requirements analyst suggests to develop the app for Android (g23), but to deliver it in twelve months (g24). These two options aim at **resolving** conflict d01, and are represented as alternative refinements r20 and r21. Assuming that no other defects are found, according to the approach provided in Sect. 2.2, we have two possible specifications: {g20, g21, g22} and {g23, g24}.

2.4 Discussion of Design Choices

Firstly, we gained a claim of completeness with respect to the defect types of our calculus by using the IEEE/ISO standards. However, there is no claim of minimality

for defect types since, e.g., unattainability constitutes a form of too strong, while ambiguity is a special case of incompleteness.

The set of refinement operators is not minimal since, e.g., **add** and **justify** modify the graph in similar ways. However, operators guide users on how to deal with defects. For example, if g is attacked as being incomplete with respect to privacy concerns, then use of **add** should introduce some privacy requirements. If, on the other hand, g is deemed unjustified, the new requirement introduced by **justify** should serve as justification for g. In short, **add** and **justify** do similar things, but for very different purposes.

It is also worth noting that, from an ontological standpoint, a statement expressing a justification (as for the **justify** refinement) should not be considered as requirement. However, in practice, stakeholders requirements often present *intermediate* requirements rather than root needs. Consider for example the following case. $\langle g_1$:"Train shall maintain a distance of $\geq$1000m from other trains"$\rangle$, given as initial requirement by a train company. Analysts may wonder where did this requirement come from, to determine whether it is defective. A justify refinement may generate $\langle g_2$:"Trains shall maintain a safe distance at all times"$\rangle$. Here, g_2 justifies g_1 because g_1 contributes to the fulfilment of g_2. This is an idea adopted from GORE, specifically *why questions* by Letier and Lamsweerde [35] where a goal is justified by the goal whose fulfilment it contributes to.

Sometimes the outputs of a refinement operation involve requirements that are also inputs, e.g., for mMissing, or even requirements higher up in the refinement chain. To avoid the introduction of cycles, CaRE supports the introduction of requirements copies that share the same description with an existing requirement but have a different identifier. For example, consider a set of requirements for a social media system that are missing privacy requirements. On applying the **add** operator, the outputs include not only the new privacy requirements but also copies of all input requirements. The copies propagate the intent of their respective original requirements but are different in an important sense: they no longer share the **missing** defect for which they were critiqued.

The **reject** defect type is different from all others in more ways than it not being refinable. Specifically, if it is applied to an initial requirement, then that requirement is unacceptable and there is no specification for the problem-at-hand. This may be a reasonable outcome in some cases. However, there is also the alternative of critiquing the unacceptable requirement as **tooStrong** and refining it to true, the null requirement. The two alternatives are different in that the first treats the requirement as unacceptable and requires a single stakeholder, while the second simply ignores the requirement, but requires the consent of all stakeholders since anyone can reject the null solution.

Still about the **tooStrong** defect, a requirement can be too strong in one sense and too weak in another. Consider the following case. The requirement $\langle g_1$:"The server shall process 1000 transactions/sec"$\rangle$ may be, at the same time, **tooStrong**(d01: "The server can't process 1000 transactions/sec"), and **tooWeak**(d02: "App will have a workload of $\geq$ 2000 transactions/sec"). Both these defects need to be addressed by adding improved requirements via refinements. For example, the former defect could be refined into $\langle g_2$:"The server shall process 800 transactions/sec"$\rangle$, whereas the latter

could be refined into the new requirements: $\langle g_3$:"The app shall run on three servers"$\rangle$, $\langle g_4$:"The system shall include a load-balancing server"$\rangle$, and $\langle g_5$:"The app shall process up to 2400 transcactions/sec"$\rangle$.

In CaRE, we do not provide an order of precedence for the application of defects and refinements types. However, in some cases, we can have 'partial' conflicts between requirements, and in these cases it can be useful to first apply the **reduce** operator, before resolving the conflict. For example, the two requirements $\langle g_1$:"The device shall be lightweight and black"$\rangle$ and $\langle g_2$:"The device shall be gray and have a large screen"$\rangle$ have a partial conflict for what concerns the color of the device. Both requirements are first critiqued as **nonAtomic**, and decomposed as $\langle g_3$:"The device shall be lightweight"$\rangle$, $\langle g_4$:"The device shall have a large screen"$\rangle$, $\langle g_5$:"The device shall be black"$\rangle$, $\langle g_6$:"The device shall be gray"$\rangle$. The conflict between g_5 and g_6 can then be resolved with one or more requirements that result from a compromise between the stakeholders.

CaRE might be criticized as too cumbersome for users compared to, say, GORE approaches. This may well be the case—we need empirical studies to determine this. However, as discussed above, CaRE is the only proposal in the RE literature for solving the requirements problem, as defined in the introduction, in its full generality.

3 Argumentation Semantics

Dung [11] introduced a formal Argumentation Framework (DAF), whose basic notions are *arguments* and *attacks* (conflicts between arguments), and where the key reasoning task is the acceptability of arguments, i.e, whether and which arguments should or should not be accepted by an intelligent agent. Sets of collectively acceptable arguments are called *extensions*[3]

The semantics of the CaRE calculus will be given in the form of a translation from a refinement graph RG into an ASPIC$^+$ *argumentation theory*, a structured variant of Dung's DAF [11]. Informally, in the translation arguments correspond to the requirements, defects and refinements of the RG. Attacks between these arguments correspond to (i) the identification of a defect in a requirement or set of requirements, and (ii) the application of a refinement to address a defect. In other words, an argument d that represents a defect in a requirement g attacks the argument representing g; similarly, an argument r that represents a refinement to address a defect d attacks the argument representing d, thus possibly restoring the acceptability of the attacked requirement. The sub-goals $g_1, ..., g_n$ proposed by the refinement in the RG willbe captured by a default implication $g_1, ..., g_n \Rightarrow r$ (See Sec.3.3 for precise details.)

Eventually, the specifications resulting from a refinement graph are computed by considering all *minimal extensions* where the initial requirements are acceptable.

The formalization of CaRE using ASPIC$^+$ is motivated by (i) the dialectic nature of requirements engineering, for which argumentation theory is a natural formal choice; (ii) the flexibility of ASPIC$^+$ being a meta-reasoning tool for reasoning over a freely chosen underlying logic, which enables us to easily consider more structured

[3] At the least, an extension S contains no arguments a and b such that a attacks b; and every argument b outside S that attacks an $a \in S$ is in turn attacked by an element of S.

RE languages, e.g. [36], in the future; (iii) the non-monotonic nature of argumentation theories, which enables extending the framework to incorporate other important features, e.g., support of conflict and dependency relations between requirements [37].

This section first introduces the formal definition of the ASPIC$^+$ structured argumentation framework, and the formal representation of a refinement graph and its well-formedness conditions. It then describes how a refinement graph is translated into an ASPIC$^+$ argumentation theory and, thereby, into a DAF. We define how this enables determining the acceptability of requirements and computing specification sets. Finally, we conclude by describing a prototype tool implementing our calculus.

3.1 Basics of Argumentation Theory

In a DAF, arguments have an abstract representation in the form of simple propositions, e.g., the argument "It is raining today, therefore I should stay home"can be represented using a simple propositional symbol a. Conflicts between arguments are given in a relation $\mathcal{D}$ over the set of arguments. For example, consider another argument b, "I have to buy food, so I must go to the store", which obviously conflicts with a. In DAF's terminology, this conflict is called an *attack*, and is represented in the form of a tuple (a, b) in $\mathcal{D}$. Given arguments and attacks, the following is one way to specify the acceptability of arguments [8]: an argument is IN (acceptable) if it is not attacked or if all its attackers are OUT (not acceptable). An argument is OUT if it is attacked by an argument that is IN. Otherwise, an argument is UNDECIDED.

Though powerful, the abstract representation of arguments in DAF makes it often less practical for modeling real-world problems. The ASPIC$^+$ framework for structured argumentation [38][4] therefore extends DAF to enable the representation of basic arguments in the form of inference rules, each having a set of premises and a conclusion. For example, argument a above can be represented using an inference rule, having a single premise "it is raining today" and a conclusion "I should stay home". One advantage of this representation is that it explicates the structure of arguments and enables the automatic construction of complex arguments by chaining inference rules. ASPIC$^+$ relies on DAF to determine *acceptability of arguments*. In particular, given an argumentation theory that includes inference rules, ASPIC$^+$ identifies the different basic and complex arguments as well as conflicts between them. Then, it constructs a DAF and uses it to determine which arguments are accepted and which are not.

3.2 Formal Description of Refinement Graphs

We start with disjoint sets of unique identifiers $\mathrm{Id_g}, \mathrm{Id_d}$ and $\mathrm{Id_r}$ for elements of the three fundamental ontological classes of Care: requirements, defects and refinements. A refinement graph RG is then tuple $\langle \mathrm{Req}, \mathrm{Defect}, \mathrm{Ref} \rangle$ where:

- $\mathrm{Req} \subseteq \mathrm{Id_g} \times \mathrm{Text}$ is a set of requirements, which includes a natural language text description.
- $\mathrm{Defect} \subseteq \mathrm{Id_d} \times \mathrm{DType} \times \mathcal{P}(\mathrm{Text}) \times \mathcal{P}(\mathrm{Id_g})$ is a set of defects. A defect includes a defect type, some natural language explanations of the defect's nature, and the set of requirements found to have the defect.

[4] In this paper, we adopt a simplified version of ASPIC$^+$.

Table 1 Mapping of Elements of Refinement Graphs to ASPIC$^+$ Argumentation Theory.

Element Type	Refinement Graph Element	ASPIC$^+$ Representation
requirement (g)	$\mathrm{Req}(g, txt)$	$\Rightarrow g$
defect (d)	$\mathrm{Defect}(d, \_, \_, G_{defective})$	$\Rightarrow d$, plus $contrary(d, g_i)$ for every $g_i \in G_{defective}$
refinement (r)	$\mathrm{Ref}(r, \_, d, G_{replace})$	$(\bigwedge_{g_i \in \mathrm{G_{replace}}} g_i) \Rightarrow r$, plus $contrary(r, d)$

– $\mathrm{Ref} \subseteq \mathrm{Id_r} \times \mathrm{RType} \times \mathrm{Id_d} \times \mathcal{P}(\mathrm{Id_g})$ is a set of refinements. A refinement has (i) an identifier; (ii) a refinement type; (iii) a defect that it aims at addressing, and (iv) a set of other requirements, which are meant to replace the defective one(s).

So, for example, from Fig. 4, we have the formal requirement $Req(g23, The\ app\ shall\ run\ on\ Android)$[5].

Henceforth, given a refinement graph RG=$\langle \mathrm{Req}, \mathrm{Defect}, \mathrm{Ref} \rangle$, the set $\mathrm{Id_{RG}}$ is used to denote $\mathrm{Id_g} \cup \mathrm{Id_d} \cup \mathrm{Id_r}$.

A refinement graph is *well-formed* iff every refinement addressing a defect matches its type, as described in Sect. 2.

3.3 Refinement Graph Semantics by Translation to Argumentation Theory

Each refinement graph RG has a corresponding ASPIC$^+$ argumentation theory representation, denoted $\mathcal{AT}(\mathrm{RG})$. An ASPIC$^+$ argumentation theory is a tuple $\langle \mathcal{L}, \mathcal{IR}, \mathrm{name} \rangle$ where

– $\mathcal{L}$ is a logical language (in our case simple propositional symbols). Instead of traditional negation, $\mathcal{L}$ is equipped with a "directional" conflict relation $contrary(q, p)$, where $contrary(q, p)$ and $contrary(p, q)$ would both be needed in order to make q behave as $\neg p$.
– $\mathcal{IR}$ is a set of defeasible inference rules of the form $\varphi_1, ..., \varphi_n \Rightarrow \varphi$, $n \geq 0$, where $\varphi_1, ..., \varphi_n, \varphi$ are from $\mathcal{L}$. In case $n = 0$, $\Rightarrow \varphi$ is equivalent to $true \Rightarrow \varphi$, and defeasibly asserts φ. The intended meaning of a defeasible rule is that if one accepts all antecedents/premises, then one must accept the consequent/conclusion unless there is sufficient reason to reject it. Defeasible rules with empty premises, of the form $\Rightarrow \varphi$, are called *assumptions*.
– $name$ is a partial function that gives names to (some) defeasible rules.

Translation of Refinement Graphs to ASPIC$^+$. The argumentation theory $\mathcal{AT}(\mathrm{RG}) = \langle \mathcal{L}_{\mathrm{RG}}, \mathcal{IR}_{\mathrm{RG}}, \mathrm{name_{RG}} \rangle$ corresponding to a refinement graph $\mathrm{RG} = \langle \mathrm{Req}, \mathrm{Defect}, \mathrm{Ref} \rangle$ is constructed using Id_{RG} as the proposition symbols $\mathcal{L}$, and the translation given in Table 1.

Thus, the above formalization represents requirements and defects as defeasible assumptions, while refinements have premises which are the requirements that the refinement introduces.

Construction of Arguments and Attacks.

ASPIC$^+$ constructs arguments that take the form of inference trees. In our case, requirements and defects are simple one-node trees, while refinements give rise to more

[5] Henceforth, we will use Req, Defect and Ref as predicates in Prolog: variables (in italics) match possible values, and underscores _ are wildcards. In logical formulas, wildcards are existentially quantified anonymous variables.

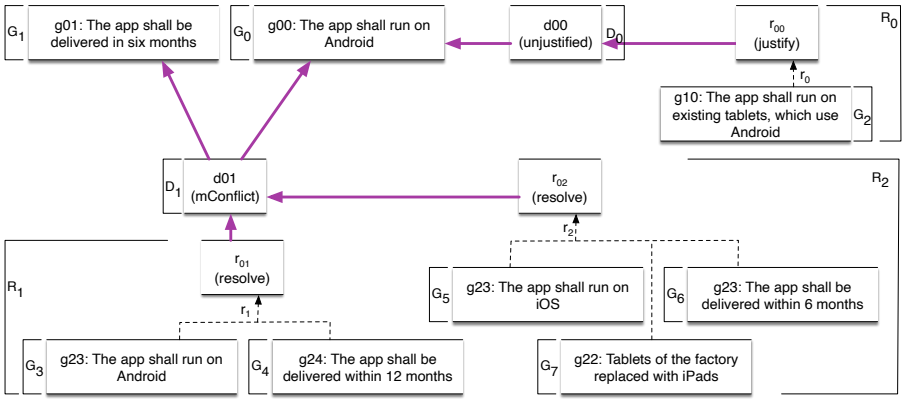

Fig. 5 Example of Construction of ASPIC$^+$ Arguments

complex trees, having as conclusion the refinement, and as premises the replacement requirements. The "contrary" relation gives rise to attacks, as described below.

Fig. 5 depicts the arguments constructed on the basis of the refinement graph shown in Fig. 4. The figure shows that all the requirements in the refinement graph correspond to arguments $\{G_0, G_1, G_2, G_3, G_4, G_5, G_6, G_7\}$ in the theory, and defects correspond to $\{D_0, D_1\}$. Refinements take the form of defeasible rules whose premises are (non-initial) requirements. These lead to inference trees where the premises are the leaves of the tree, as in $\{R_0, R_1, R_2\}$. Notice the structure of every argument: it includes a set of premises, a conclusion, and a set of defeasible rules. For example, the premises of argument R_2 are G_5, G_6 and G_7, its conclusion is r_{02}, and it has a single defeasible rule r_2.

Identification of Attacks. Given two ASPIC$^+$ arguments A and B, A attacks B if the conclusion of A conflicts with the conclusion or premises of the rule for B. Note that formula ϕ conflicts with ψ if it contradicts it: $contrary(\phi, \psi)$.

Accordingly, defects attack requirements that they point to, whereas refinements attack defects that they address. So, for example, Fig. 5 shows that defect D_0 attacks requirement G_0, and then the refinement R_0 attacks the defect D_0. Though not shown in Fig. 5, a defect can attack ("rebut") not just requirement g, but also attack ("undermine") any refinement that uses g.

Construction of DAFs. The purpose of argument construction and attack identification in ASPIC$^+$ is to enable the construction of a DAF.

We present the construction process by example here (for details, see [38]): starting from the theory above, one obtains a DAF that can be represented graphically as in Fig. 6, where nodes represent arguments, and edges represent attacks. One can easily see in the graph how arguments $\{D_0, D_1\}$, representing defects, attack arguments $\{G_0, G_1\}$, representing requirements. Similarly, arguments $\{R_0, R_1, R_2\}$, denoting refinements, attack arguments $\{D_0, D_1\}$, denoting defects.

Computation of DAF Extensions The computation of the extensions of a DAF enables determining which arguments should be accepted and which should not. The computation of extensions is based on the following concepts and definitions.

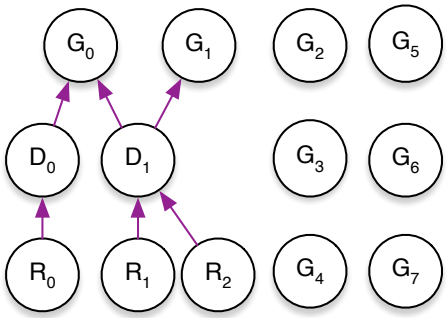

Fig. 6 DAF Example

- A set A of arguments is *conflict-free* if it does not include two arguments that attack each other.
- An argument a is *acceptable* w.r.t. to a set of arguments A iff whenever a is attacked by an argument b then b must be attacked by some element in A.
- A set of arguments A is *admissible* iff A is conflict-free and every argument $a \in A$ is acceptable w.r.t. A.
- A set of arguments is *complete* iff if is admissible and includes every argument a that is acceptable w.r.t. to it.

In this paper, we are interested in the computation of the complete extensions. In the previous example, the only complete extension is the set $\{G_0, G_1, G_2, G_3, G_4, G_5, G_6, G_7, R_0, R_1, R_2\}$. In general, if a DAF graph is acyclic then it is guaranteed to have a single complete extension.

In general, the *acceptable requirements* in a refinement graph RG, denoted by $AR(\mathcal{AT}(\mathrm{RG}))$, will be the set of requirements appearing in the conclusions of the arguments of its complete extension.

After the identification of acceptable arguments, we determine acceptable requirements by checking the ones that appear as conclusions of acceptable arguments. Thus, we determine that all the requirements $\{G_0$,...,$G_7\}$ are acceptable since they are the conclusions of such arguments .

3.4 Identification of Specification Sets

The acceptability of requirements only indicates that either they are free of defects or their defects have been addressed. To determine the minimal sets of requirements necessary to make the initial requirements acceptable, we compute the minimal specification sets. In the following, suppose we are given a specific requirements graph RG = $\langle$Req, Defect, Ref$\rangle$.

The *initial requirements* InitR are those that are not introduced by a refinement. Formally, InitR=$\{g \mid \neg\exists RF, id \ .\ \mathrm{Ref}(\_, \_, \_, RF) \ \wedge \ g \in RF \ \}$, where RF is a variable ranging over sets of requirements in the refinement graph, more precisely $RF \in \mathcal{P}(\mathrm{Id_g})$[6].

[6] Note that we adopt the Prolog notation of using symbols starting with upper-case letters to represent variables and the underscore symbol _ to denote an anonymous variable.

The *specification elements* SpecE are leaves of a refinement graph that have $isFinal$ true, and are therefore "acceptable". As such, they have no defects that are unaccounted for.

Note that SpecE contains, among others, what would normally be called the functional requirements. If a functional requirement (e.g., compute the square root) is faulted (e.g., for ambiguity) then the result of refining it (e.g., compute the positive square root) replaces it in SpecE. So functional requirements require no special treatment in CaRE.

Let a *minimal set of requirements* be a (minimal) subset of the requirements Req that lead to the acceptance of the initial requirements. Formally, it is one of the subsets of RR which is minimal w.r.t. set inclusion:

$\mathrm{RR} = \{R' \mid RG' = \langle R', \mathrm{Fault}, \mathrm{Ref}\rangle \wedge R' \subseteq \mathrm{Req} \wedge \mathrm{InitR} \subseteq AR(\mathcal{AT}(RG'))\}$

Intuitively, the set RR is the set of all subsets of the requirements proposed during refinements that lead to the acceptance of the initial requirements. In the running example, the sets $\{G_3, G_4, G_5\}$, $\{G_2, G_3, G_4, G_5\}$, and $\{G_2, G_3, G_4, G_5, G_6\}$ represent some of the elements of RR. The minimal requirements sets are $\{G_2, G_3, G_4\}$ and $\{G_2, G_5, G_6, G_7\}$. Finally, the *specification sets*, SS, are identified by taking the intersection of *specification elements* and *minimal requirements*, i.e., $\mathrm{SS} = \{\mathrm{S} \mid \exists R.\mathrm{R} \in \mathrm{RR}\,, \mathrm{S} = (\mathrm{R} \cap \mathrm{SpecE})\}$. In the running example, the sets $\{G_3, G_4\}$ and $\{G_5, G_6, G_7\}$ represent the only specification sets.

3.5 Tool Description

We have implemented a prototype tool of the calculus. The tool aims at helping requirements engineers to systematically refine, negotiate, and document the requirements refinement process (in the form of a refinement graph). The tool also provides reasoning support by determining acceptability of requirements and computing minimal specifications. A brief description of the tool is presented below. The tool, as well a description of the examples in this paper and use instructions, can be downloaded at [12] (requires Java SE Development Kit 9 to run). The tool's input is a textual description of a refinement graph—a GUI is left as future work. An "Argumentation Theory Generator" module then generates an ASPIC$^+$ argumentation theory for every possible configuration of requirements. A *requirements configuration* is a subset of the requirements that could lead to the acceptance of the initial requirements. On the basis of these argumentation theories, an "ASPIC$^+$ module" identifies the ASPIC$^+$ arguments, attacks, and generates a Dung Argumentation Framework (DAF). A "DAF module" then determines acceptability of abstract arguments by computing complete extensions of the DAF. Finally, a "Compute Minimal Specifications" module stores all (subsets of) requirements (RR) that make the initial requirements acceptable and determines minimal specification sets (SS) by taking the intersection of specification elements and minimal requirements (as explained in Sec. 3.4).

4 A (sometimes) Complementary Semantics using Horn Logic

An argumentation-based semantics is the most appropriate for CaRE because the entire calculus is dialectic: a cycle of pointing out defects and then arguing against them

through refinements. However, in the past most requirements modeling languages were given semantics using some form of logic based on predicates/propositions combined with the usual logical connectives, and accompanied by special inference rules: in addition to standard First Order Logic (e.g., for RML [21]), requirements which permitted inconsistencies (as does CaRE) were formalized using some form of para-consistent logic; this includes default-logic (Zowghi and Offen [53], Ghose [20]), labeled quasi-classical logic [27], and maximal consistent subsets [15]. The last was implemented using Assumption-based Truth Maintenance Systems, which perform abductive reasoning.

In this section, we show how to translate a refinement graph RG into a propositional Horn logic theory $\mathcal{HT}(RG)$, and define a solution to the requirements problem as corresponding to performing abduction over a version of $\mathcal{HT}(RG)$. The important property of this new definition of "solution" is that it can be *proven to be equivalent* to the specification sets defined earlier in Sec. 3.4, based on argumentation theory, though the proof only holds in a special case.

The importance of providing such *"consistent complementary semantics"* is that it increases our confidence in both, since we approach the problem from two different viewpoints, each of which might let us overlook (different) things. This was first used in Mathematical Logic, where soundness and completeness results connected Tarskian semantics with proof theory; Hoare & Lauer [26] were the first to prove consistent complimentary formal semantics in the field of Programming Languages, where this paradigm continues till this day.

4.1 Translation of Refinement Graphs into Propositional Logic.

The intuition behind the translation lies in the observation that a requirement without any defects is always acceptable; and if the requirement has defects, it can only be deemed acceptable if all of such defects have been addressed by refinements, whose subgoals are themselves acceptable. So basically, we do not need propositions for refinements.

For example, consider the CaRE graph in Fig. 4. The following shows the translation of its left portion in view of the above intuitions. (For increased readability, we have used more mnemonic identifiers, and added surrounding predicate names $Acceptable(\_)$ and $Addressed(\_)$ for requirements and defects respectively.). The first three axioms indicate requirements without defects; the next one shows one with 2 defects; and the last two rules show how defects have been addressed.

$$
\begin{array}{lcl}
Acceptable(g10.forAllAndroidTblets) & \leftarrow & true \\
Acceptable(g23.runOnAndroid) & \leftarrow & true \\
Acceptable(g24.deliveredIn12Mo) & \leftarrow & true \\
 & & \\
Acceptable(g00.runOnAndroid) & \leftarrow & Addressed(d00.unjustified) \wedge \\
 & & \quad Addressed(d01.cannotDeliverIn6mo) \\
 & & \\
Addressed(d00.unjustified) & \leftarrow & Acceptable(g10.forAllAndroidTablets) \\
Addressed(d01.cannotDeliverIn6mo) & \leftarrow & Acceptable(g23.runOnAndroid) \wedge \\
 & & \quad Acceptable(g24.deliveredIn12Mo)
\end{array}
$$

In general, given a formal description of a refinement graph RG, the logical theory $\mathcal{HT}(RG)$ will have as propositional symbols the set of identifiers for requirements

Table 2 Mapping of Elements of Refinement Graphs to Propositional Logic

Element Type	Refinement Graph Element	Horn representation
requirement (g)	$\mathrm{Req}(g, txt)$	$g \leftarrow true$
defect (d)	$\mathrm{Defect}(d, \_, \_, G_{defective})$	Replace $g_i \leftarrow \varphi$ with $g_i \leftarrow \varphi \wedge d$ for every $g_i \in G_{defective}$
refinement (r)	$\mathrm{Ref}(r, \_, d, G_{replace})$	$d \leftarrow \bigwedge_{g_i \in \mathrm{G_{replace}}} g_i$

and defects in RG. For each element of RG, we add or modify a rule in the theory $\mathcal{HT}(RG)$ being built, according to Table 2. Essentially, when a defect of requirement r appears in RG, it is conjoined to the antecedent of the implication for r to indicate that all defects of a requirement must be handled before it can be considered to be addressed. As an aside, we note that the above construction could no longer be carried out in the case when CaRE would be extended to allow refinements themselves to be attacked as defective by other agents — a quite reasonable scenario (see Fig.4 in [50]) which we plan to consider in future work.

The following theorem connects the two semantics in the case of requirement graphs that are trees

Theorem 1 *Given a requirements graph RG that is a tree/forest, the set of requirements derivable from $\mathcal{HT}(RG)$ is identical to the set of requirements appearing in the unique complete and grounded extension of $\mathcal{AT}(RG)$.*

The theorem can be extended to acyclic requirement graphs by duplicating nodes, as illustrated in [47] for example.

The proof of this result can be found at [12], and relies on generalizing refinement graphs to allow defect (resp. refinement) nodes to exist without being connected to the requirement (resp. defect) nodes they target (resp. address), and then proceeds by a form of induction on the number of target and address edges.

4.2 Requirements Problem Solutions and Abduction

As stated in Sec. 3.4, solutions to the requirements problem in CaRE correspond to minimal subsets of leaf requirements (which can be viewed as specifications) that make the initial requirements $InitR$ acceptable.

Note that leaf requirements g, which have no defects, can be identified in $\mathcal{HT}(RG)$ by the fact that their rules have the form $g \leftarrow true$. If we define $Abd(\mathcal{HT}(RG))$ as the subset of $\mathcal{HT}(RG)$ that omits such rules, then abduction identifies exactly such minimal subsets for "explaining" $InitR$. More formally (cf. [42])

Definition 1 Given a consistent (Horn) theory T (called the *background theory*) over alphabet Σ, a formula ξ, (called the *query*), and a set of literals $A \subseteq \Sigma$ (called the *abducibles*), an *explanation of ξ with respect to A from T* is a minimal set of literals $E \subseteq A$, such that (i) $\Sigma \cup E \models \xi$ and (ii) $\Sigma \cup E$ is satisfiable.

In the case when Theorem 1 holds, the problem of finding the specification sets SS of RG is then identical to the problem of finding explanations SS of $\bigwedge_{id_{g_i} \in InitR} id_{g_i}$ with respect to $Leafs(RG)$ from $Abd(\mathcal{HT}(RG))$.

Although there may be exponentially many such explanations SS, the problem of finding each one is actually in PTIME for definite Horn theories [42] such as ours – ones where there are no rules of the form $\bot \leftarrow \varphi$. These solutions can be found using an Assumption-based Truth Maintenance System (ATMS) [10].

5 Application Scenario

We showcase the proposed calculus on an example of a train control system, inspired from the ERTMS/ETCS Level 3 moving block system [16]. The ERTMS/ETCS system is the European standard for train control and management. In its deployed Level 2 implementations, the location of trains is identified by means of fixed devices, called *balises*, which are positioned along the track. The balise communicates its location when the train passes over it, so that the train has an accurate, yet *intermittent*, information about its position, which is shared with the central control system. In this way, the central system can provide train separation by allocating fixed segments of line, called *block sections*, to each single train. The length of the segment is established based on the distance between balises. With the ERTMS/ETCS Level 3, currently under study, the location of trains is identified by means of satellite navigation, which enables *continuous* train positioning. If the central system can identify the exact location of each train, more trains can be safely routed along the same track. This concept is known as *moving block*, since the block section, i.e., the segment of line allocated to a train, can vary in a continuous and dynamic way, depending on the position of the preceding train.

In our scenario, we consider a fictional negotiation process between the stakeholders involved in the definition of the requirements of ERTMS/ETCS Level 3, showing how our approach could be applied. Although based on a simplification of the problem, we believe that the example is representative of issues that may emerge in a real-world requirements negotiation scenario. We consider only two subjects involved in the negotiation, the stakeholder (S), representative of the different parties who have an interest in the development of the system, and the requirements analyst (A), representative of the party who will develop the system. Through the example, we discuss evolution of the refinement graph by depicting results obtained using the tool described in Section 3.5.

The basic graphical elements of the refinement graph are in Fig. 3. Fig. 7 reports the refinement graph for the scenario. Numbers in boxes associated to requirements—not currently part of the notation—identify the step in which a certain refinement took place. To facilitate the reader, we will also report part of the intermediate steps of the refinement procedure.

Step 0: The initial requirements proposed by S are:

- g00: The system shall ensure safety distance between trains
- g01: The distance between trains shall be minimal
- g02: The train location shall be identified through satellite navigation

Step 1: A argues that the provided requirements are not associated to justifications (unjustified defects d01, d02, d03), and S provides them for each requirement (justify refinements r10, r11, r12). Fig. 8 reports the refinement graph at this step.

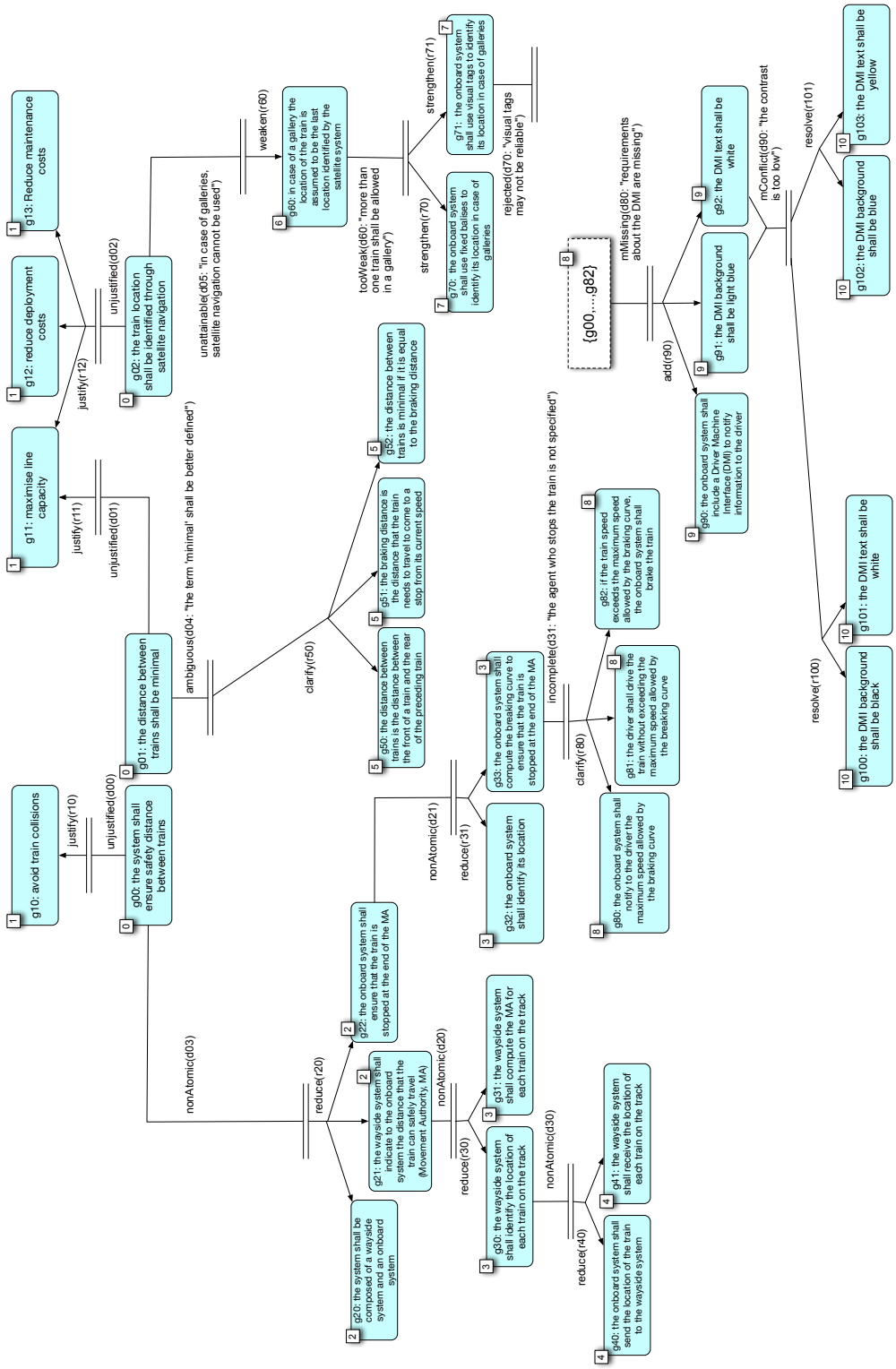

Fig. 7 Refinement Graph for the ERTMS/ETCS Level 3 application scenario.

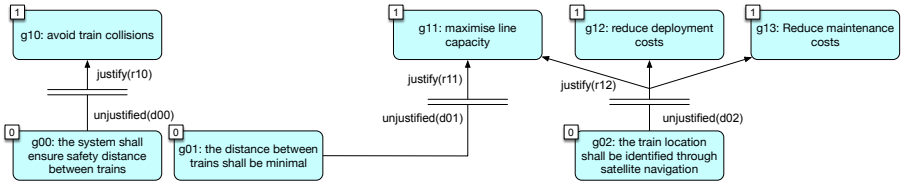

Fig. 8 Refinement Graph for the ERTMS/ETCS Level 3 application scenario - Step 1.

- g00 → g10: avoid train collisions
- g01 → g11: maximise line capacity
- g02 → g12: reduce deployment costs
- g02 → g13: reduce maintenance costs
- g02 → g11: maximise line capacity

Step 2: g00 is too abstract (nonAtomic(d03)), and S performs a first attempt to decompose it to come towards more refined requirements (reduce(r20)).

- g00 → g20: the system shall be composed of a wayside system and an onboard system
- g00 → g21: the wayside system shall indicate to the onboard system the distance that the train can safely travel (Movement Authority, MA)
- g00 → g22: the onboard system shall ensure that the train is stopped at the end of the MA

Step 3: g21 and g22 are still too abstract, and further decomposition is performed.

- g21 → g30: the wayside system shall identify the location of each train on the track
- g21 → g31: the wayside system shall compute the MA for each train on the track
- g22 → g32: the onboard system shall identify its location
- g22 → g33: the onboard system shall compute the breaking curve to ensure that the train is stopped at the end of the MA

Step 4: Further refinements are the following. An excerpt of the refinement graph at this step is reported in Fig. 9. The excerpt does not report the refinements stemming from g01 as these did not further evolve at this stage.

- g30 → g40 the onboard system shall send the location of the train to the wayside system
- g30 → g41 the wayside system shall receive the location of each train on the track

Step 5: A argues that the term “minimal”, used in g01, is vague (ambiguous(d04)), so S clarifies it (clarify(r50)).

- g01 → g50: the distance between trains is the distance between the front of a train and the rear of the preceding train
- g01 → g51: the breaking distance is the distance that the train needs to travel to come to a stop from its current speed
- g01 → g52: the distance between trains is minimal if it is equal to the breaking distance

Step 6: A argues that satellite navigation required by g02 may not work in case of galleries, and this makes the requirement unattainable. A specific requirement for galleries is defined that weakens the original requirement (weaken(r60)).

- g02 → g60: in case of a gallery the location of the train is assumed to be the last location identified by the satellite system

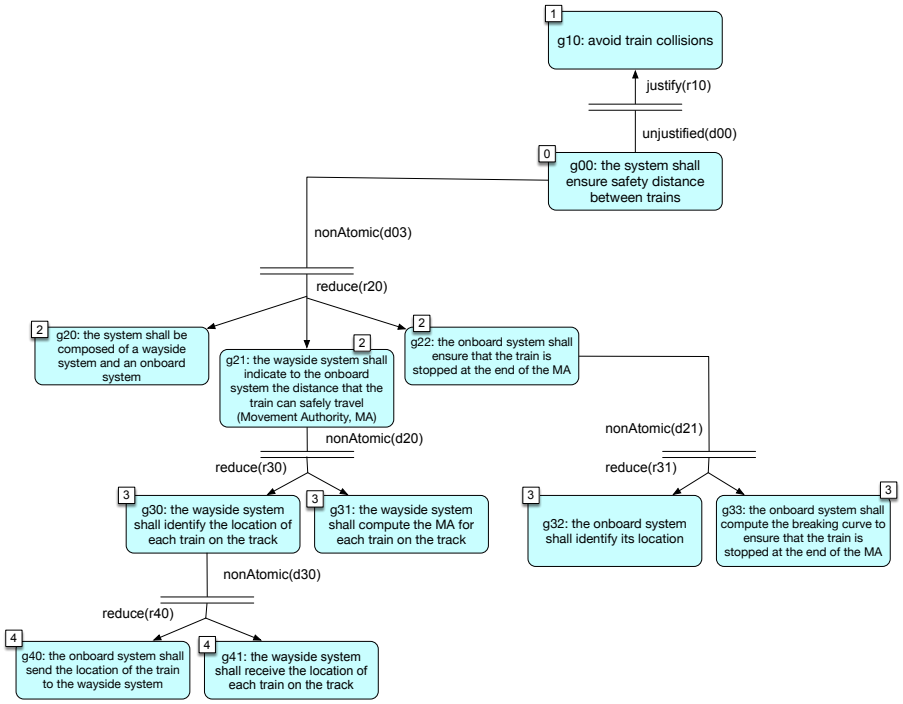

Fig. 9 Refinement Graph for the ERTMS/ETCS Level 3 application scenario - Step 4 (Excerpt).

Step 7: g60 is not acceptable for S, since it implies that only one train is allowed to move in a gallery (the requirement is considered tooWeak). To strengthen this requirement, A proposes to either use the traditional fixed balise system to locate the train in galleries (strengthen(r70)), or to use visual tags placed along the gallery (strengthen(r71)), which should be detected by a camera on the train. This second option is rejected, as this system may not be sufficiently reliable. the relevant excerpt of the refinement graph at this step is reported in Fig. 10. We do not report the refinements stemming from g00 as these did not further evolve at this stage

- g60 → g70: the onboard system shall use fixed balises to identify its location in case of galleries
- g60→ g71: the onboard system shall use visual tags to identify its location in case of galleries

Step 8: By reviewing the requirements, A notices that g33 does not specify *who* is stopping the train. The requirement is deemed incomplete (incomplete(d31)). S specifies that the driver is in charge of driving and stopping the train, while the onboard system monitors that the breaking curve is respected. If the driver violates the braking curve, the onboard system brakes the train. Hence, the requirement is clarified as follows:

- g33 → g80: the onboard system shall notify to the driver the maximum speed as allowed by the braking curve to stop at the end of the MA
- g33 → g81: the driver shall drive the train without exceeding the maximum speed allowed by the braking curve
- g33 → g82: if the train speed exceeds the maximum speed allowed by the braking curve, the onboard system shall brake the train

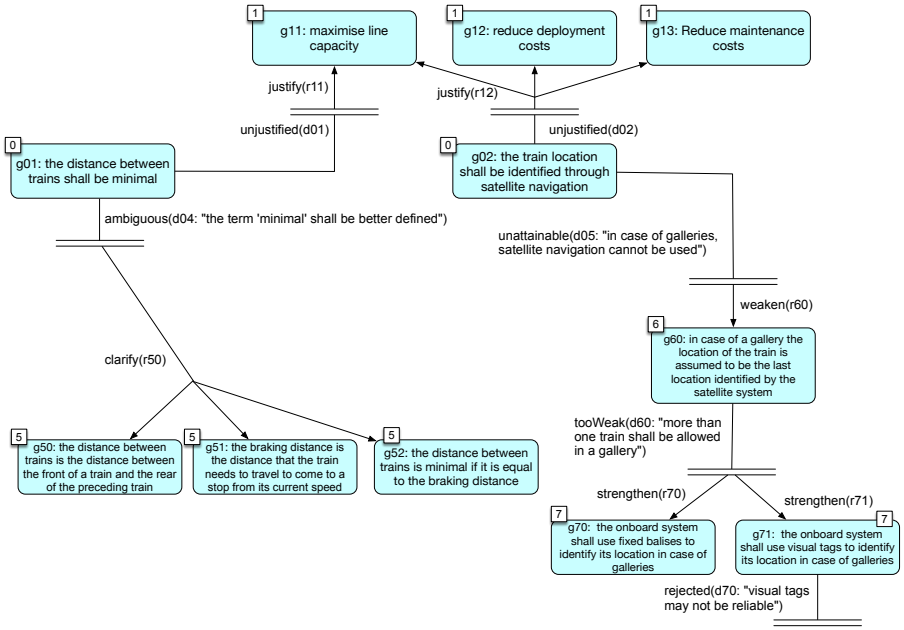

Fig. 10 Refinement Graph for the ERTMS/ETCS Level 3 application scenario - Step 7 (Excerpt).

Step 9: A understands that there is a human agent in the system (the train driver), but no requirement is specified concerning the communication with this agent. The requirements set is considered incomplete (mMissing(d80)). Therefore, A solicits S to discuss driver-related requirements. The following requirements are introduced (add(r90)):

- g90: the onboard system shall include a Driver Machine Interface (DMI) to notify information to the driver
- g91: the DMI background shall be light blue
- g92: the DMI text shall be white

Step 10: A observes that the contrast may be insufficient if the background is light blue and the text is white (mConflict(d90)). Two options are proposed to resolve the conflict.

- g91, g92 $\rightarrow$ g100, g101: the DMI background shall be black, the DMI text shall be white
- g91, g92 $\rightarrow$ g102, g103: the DMI brackgroun shall be blue, the DMI text shall be yellow

We run the tool on the previous example using a MacBook pro, with a 2.2 GHz Intel i7 processor and 32 GB of DDR RAM. The tool read the refinement graph and identified its initial and specification nodes in approximately 23ms. The computation of the minimal specifications took around 280ms producing the two specifications: $\{$g20, g31, g32, g40, g41, g50, g51, g52, g70, g80, g81, g82, g90, g100, g101$\}$, and $\{$g20, g31, g32, g40, g41, g50, g51, g52, g70, g80, g81, g82, g90, g102, g103$\}$.

This scenario can be run using the tool that is made available at the following DOI: 10.5281/zenodo.3856402.

6 Related Work

RE Models and Refinement The use of models to support the RE refinement process has been extensively studied. Early contributions include the SADT methodology [41] and its extension RML[22], where activity blocks are hierarchically decomposed into sub-blocks until atomic and operationalisable entities are obtained. Similarly, basic Goal-Oriented Requirements Engineering (GORE) [9,49] relies mainly on and-or decomposition of high-level goals until all implementable atomic requirements, domain assumptions and properties needed to satisfy the high-level goals are identified. Around GORE, a whole family of methods and tools was developed, each with a different focus. For instance, KAOS [34,46] is a basic GORE model augmented with obstacles to model issues that can prevent satisfaction of a goal [45]; support of quantitative aspects and uncertainty are studied in [24] and [6,7] respectively. The i* modelling framework [48] and its related software engineering method Tropos [5] focus on modeling of social aspects and analysis. None of these frameworks accounts for the argumentative and dialectic aspects of the RE process. In contrast, CaRE makes the various requirement defects explicit first-class entities in the model and provides a comprehensive set of refinement operators to address each. This is a more comprehensive and arguably a more natural modeling of the actual refinement process, allowing a better documentation of its evolution over time. Furthermore, change management in CaRE is simpler than in these frameworks since change can always be made through additions to the model, as opposed to a revision or deletion of already existing elements. In other words, building of models, namely CaRE refinement graphs, is monotonic.

Argumentation and RE Argumentation has been often used for modeling and reasoning about RE artifacts and processes. One of the earliest works that used argumentation in RE was [23], where *security satisfaction arguments*, based on *Toulmin* arguments [44], are used to convince a reader of the satisfaction of security requirements in a system. This work was extended in [19] with a risk assessment method called (RISA), which enables identification of rebuttals and mitigations needed to satisfy security requirements. In [29], a five-step iterative process is defined to systematically establish compliance of system requirements with law through discussions among stakeholders. In [14], argumentation is used to model requirements elicitation and explain common cases of ambiguity. Dung's abstract argumentation framework is used in Bagheri and Ensan [3] to model interaction and inconsistencies between requirement statements, and to rank possible (consistent) subsets of requirement statements. These contributions have a different focus than ours as they target requirements elicitation [14], assessment [23,19], regulatory compliance [29], security [23,19], and requirements conflicts [3,39].

Closer to the spirit of our work are the Goal Argumentation Method (GAM) by Jureta et al. [31,32], the RationalGLR method by van Zee et al. [50,52,51], Mirbel & Villeta [37], and the Desirée proposal by Li et al. [36]. In Jureta et al. [31, 32], the GAM framework supports definition of goal models and representation of information exchanged in a discussion about the relative validity of an RE artifact, and defines algorithms to label information produced as *accepted*, *inconclusive* or

rejected. RationalGRL [50,52,51] captures not only traditional GORE model refinement, but also arguments about design decisions (e.g., "This refinement should be OR rather than AND"), and the rationale behind them. It also proposes the use of argumentation patterns to point out defects in goal models. Mirbel & Villeta [37], given a traditional GORE model extended with *Conflict*, *Requires*, and *Equivalence* relations, relies on Argumentation Theory to identify possible sets of requirements that are consistent. The main feature distinguishing our work from all of the above is its scope and approach: CaRE aims at covering the complete RE refinement process by providing a comprehensive *calculus* for deriving specifications from stakeholder requirements. Thus, CaRE supports the set of defects identified in IEEE standards, as well as refinement operators needed for addressing each specific defect type.

The only other work we know of that offers a refinement calculus for the requirements problem is the Desirée proposal [36], which generalizes GORE approaches with a rich set of operators for refinement and operationalization. The main differences between CaRE and Desirée are that CaRE (a) includes defects and defect types in its ontology, which Desirée does not, (b) casts the refinement process as a dialectic argument among stakeholders, and (c) gives a formal semantics of what it means for S to satisfy R based on Argumentation Theory.

Finally, we mention two papers that foresaw the importance of processes for generating specifications. Finkelstein et al. [18] argue that such processes are inherently ill-understood and require formal models with precise, unambiguous and analyzable semantics. The paper also argues that conventional deductive logics are not adequate for the semantics of these processes and suggests that non-standard logics should be tried instead. Among the non-standard logics discussed are dialogue logics, since Dung's Argumentation Theory was proposed later. Black et al. [4] also focus on the process of generating specifications and cast such processes as agent dialogues consisting of moves that participating agents make about elements of a specification. This work adopts a very different ontology from CaRE, missing the notions of requirement, defect and refinement. It also leaves open the question of semantics for process enactment.

7 Conclusion

This paper presents a novel calculus for RE to support the derivation of a specifications from initial stakeholder requirements through a dialectic process founded on the notions of requirement/goal, defect and refinement. One important advantage of our proposal is that it covers *the full range of defects* recognized in RE standards, and offers refinement operators to address each defect type. It also makes the stakeholders—or their representative in case of large numbers—active participants in the refinement process, as opposed to traditional approaches where typically only requirements analysts participate in the analysis of stakeholder requirements.

CaRE refinement graphs capture a comprehensive view of RE process enactments and provide excellent support for RE documentation, traceability, and change management since new defects or refinements can be added to the graph monotonically, without a need to revise any of its previous elements.

The semantics of the calculus is given in terms of Argumentation Theory, by defining a mapping from elements of refinement graphs into constructs of the ASPIC$^+$ argumentation framework. In our proposal, the notion of requirements *satisfaction* or *fulfillment* typical of earlier approaches, is replaced by the weaker notion of *acceptability*. Our contributions include a Java implementation of a prototype tool for the calculus and a realistic application scenario, as preliminary evidence of the soundness and viability of our proposal. We also offer a complementary semantics for CaRE based on a more traditional translation of refinement graphs into Horn Logic, whereby specifications are derived from initial requirements through abduction.

We are aware of a number of limitations of the current work: many are issues addressed for the KAOS GORE technique by van Lamsweerde [33] (e.g., integration with use-cases, industrial strength tool support and UI design, full empirical evaluation); others are more general RE issues, such as accounting for stakeholder preferences and negotiations. At a lower level, we want to look at the effect of various argumentation semantics, and for a systematic approach to propose refinement operators for each step of the refinement process.

Thus, future work includes a preliminary assessment of CaRE on an industrial case-study, and a consolidation assessment of domain experts using CaRE. Other future work includes adding further aspects of GORE ontologies (e.g., soft-goals, agents), relations (*equivalence*, *requires*, *conflicts*, etc), and global consistency conditions on requirements graphs (e.g., what are *valid* refinements and attaching logical meaning to "weaken", "strengthen", "conflict", esp. once we interpret requirements as formulas that are part of a domain theory). We also plan to improve the prototype implementation tool through development of a Graphical User Interface, and to study the use of CaRE for training and education of requirements engineers.

References

1. IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830-1998 pp. 1–40 (1998)
2. Iso/iec/ieee international standard - systems and software engineering – life cycle processes – requirements engineering. ISO/IEC/IEEE 29148:2011(E) (2011)
3. Bagheri, E., Ensan, F.: Consolidating Multiple Requirement Specifications Through Argumentation. ACM SAC pp. 659–666 (2011)
4. Black, E., McBurney, P., Zschaler, S.: Towards agent dialogue as a tool for capturing software design discussions. In: International Workshop on Theorie and Applications of Formal Argumentation, pp. 95–110. Springer (2013)
5. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems **8**(3), 203–236 (2004)
6. Cailliau, A., van Lamsweerde, A.: Assessing requirements-related risks through probabilistic goals and obstacles. Requirements Engineering **18**(2), 129–146 (2013)
7. Cailliau, A., van Lamsweerde, A.: Runtime monitoring and resolution of probabilistic obstacles to system goals. ACM Trans. Auton. Adapt. Syst. **14**(1), 3:1–3:40 (2019). DOI 10.1145/3337800. URL https://doi.org/10.1145/3337800
8. Caminada, M.: On the issue of reinstatement in argumentation. In: M.F. et al (ed.) Logics in Artificial Intelligence, pp. 111–123. Springer (2006)
9. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Sci. Comput. Program. **20**(1-2), 3–50 (1993)
10. De Kleer, J.: An assumption-based tms. Artificial intelligence **28**(2), 127–162 (1986)

11. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. AI Journal **77**(2), 321–357 (1995)
12. ElRakaiby, Y., Borgida, A., Ferrari, A., Mylopoulos, J.: CaRE: A Refinement Calculus for Requirements Engineering based on Argumentation Theory (Proofs and Tool) (2020). DOI 10.5281/zenodo.4156216. URL https://doi.org/10.5281/zenodo.4156216
13. Elrakaiby, Y., Ferrari, A., Mylopoulos, J.: Care: A refinement calculus for requirements engineering based on argumentation semantics. In: 2018 IEEE 26th International Requirements Engineering Conference (RE), pp. 364–369 (2018). DOI 10.1109/RE.2018.00-24
14. Elrakaiby, Y., Ferrari, A., Spoletini, P., Gnesi, S., Nuseibeh, B.: Using argumentation to explain ambiguity in requirements elicitation interviews. In: RE'17, pp. 51–60. IEEE (2017)
15. Ernst, N.A., Borgida, A., Mylopoulos, J., Jureta, I.: Agile requirements evolution via paraconsistent reasoning. In: Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7328, pp. 382–397. Springer (2012)
16. European Union Agency for Railways: ERTMS/ETCS System Requirements Specification 3.4.0. http://www.era.europa.eu/Document-Register/Pages/Set-2-System-Requirements-Specification.aspx (2016)
17. Fernández, D.M., Wagner, S., Kalinowski, M., Felderer, M., Mafra, P., Vetrò, A., Conte, T., Christiansson, M.T., Greer, D., Lassenius, C., et al.: Naming the pain in requirements engineering. Empirical software engineering **22**(5), 2298–2338 (2017)
18. Finkelstein, A.: Modeling the software process:"not waving but drowning" (panel session) representation schemes for modelling software development. In: Proceedings of the 11th international conference on Software engineering, pp. 402–404 (1989)
19. Franqueira, V.N.L., Tun, T.T., Yu, Y., Wieringa, R., Nuseibeh, B.: Risk and argument: A risk-based argumentation method for practical security. In: RE 2011, pp. 239–248 (2011)
20. Ghose, A.: Formal tools for managing inconsistency and change in RE. In: Int. Workshop on Software Specification and Design, pp. 171–181 (2000). URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=891138
21. Greenspan, S.J., Borgida, A., Mylopoulos, J.: A requirements modeling language and its logic. Information Systems **11**(1), 9–23 (1986)
22. Greenspan, S.J., Mylopoulos, J., Borgida, A.: Capturing more world knowledge in the requirements specification. In: Proceedings of the 6th International Conference on Software Engineering, ICSE '82, p. 225–234. IEEE Computer Society Press, Washington, DC, USA (1982)
23. Haley, C.B., Laney, R., Moffett, J.D., Nuseibeh, B.: Security requirements engineering: A framework for representation and analysis. TSE **34**(1), 133–153 (2008)
24. Heaven, W., Letier, E.: Simulating and optimising design decisions in quantitative goal models. In: 2011 IEEE 19th International Requirements Engineering Conference, pp. 79–88. IEEE (2011)
25. Hegel, G.W.F.: Ph´anomenologie des Geistes (1807)
26. Hoare, C.A.R., Lauer, P.E.: Consistent and complementary formal theories of the semantics of programming languages. Acta Informatica **3**(2), 135–153 (1974)
27. Hunter, A., Nuseibeh, B.: Analysing inconsistent specifications. In: RE, pp. 78–86 (1997). DOI 10.1109/ISRE.1997.566844. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=566844
28. Hunter, A., Nuseibeh, B.: Managing inconsistent specifications: reasoning, analysis, and action. ACM Trans. Softw. Eng. Methodol. **7**(4), 335–367 (1998)
29. Ingolfo, S., Siena, A., Mylopoulos, J., Susi, A., Perini, A.: Arguing regulatory compliance of software requirements. DKE **87**, 279–296 (2013)
30. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: ICSE'95, pp. 15–15. IEEE (1995)
31. Jureta, I.J., Faulkner, S., Schobbens, P.Y.: Clear justification of modeling decisions for goal-oriented requirements engineering. Requirements Engineering **13**(2), 87 (2008)
32. Jureta, I.J., Mylopoulos, J., Faulkner, S.: Analysis of multi-party agreement in requirements validation. In: RE'09, pp. 57–66 (2009)
33. van Lamsweerde, A.: Goal-oriented requirements enginering: A roundtrip from research to practice. In: RE'04), pp. 4–7 (2004)
34. Lamsweerde, A.v.: Requirements Engineering: From System Goals to UML Models to Software Specifications, 10 edn. John Wiley & Sons, Chichester, UK (2009)
35. Letier, E., van Lamsweerde, A.: Kaos in action: the bart system. In: IFIP WG2, vol. 9 (2000)

36. Li, F.L., Horkoff, J., Borgida, A., Guizzardi, G., Liu, L., Mylopoulos, J.: From stakeholder requirements to formal specifications through refinement. In: Conf. on Requirements Engineering: Foundation for Software Quality (REFSQ'15), pp. 164–180. Springer (2015)
37. Mirbel, I., Villata, S.: Enhancing Goal-based Requirements Consistency: an Argumentation-based Approach. In: Int. Work. Comput. Log. Multi-Agent Syst., pp. 110–127 (2012)
38. Modgil, S., Prakken, H.: The ASPIC+ framework for structured argumentation: a tutorial. Argument Comput. **5**, 31–62 (2014)
39. Murukannaiah, P.K., Kalia, A.K., Telangy, P.R., Singh, M.P.: Resolving goal conflicts via argumentation-based analysis of competing hypotheses. In: Proc. RE'15, pp. 156–165 (2015)
40. Potts, C., Takahashi, K., Anton, A.I.: Inquiry-based requirements analysis. IEEE software **11**(2), 21–32 (1994)
41. Ross, D.T.: Structured analysis (sa): A language for communicating ideas. IEEE TSE (1), 16–34 (1977)
42. Selman, B., Levesque, H.J.: Abductive and default asoning: A computational core (1990)
43. Tjong, S.F., Berry, D.M.: The design of SREE: a prototype potential ambiguity finder for requirements specifications and lessons learned. In: REFSQ'13, pp. 80–95. Springer (2013)
44. Toulmin, S.E.: The uses of argument. Cambridge university press (2003)
45. van Lamsweerde, A.: Handling obstacles in goal-oriented requirements engineering. IEEE TSE **26**(10), 978–1005 (2000)
46. Van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: Proceedings fifth IEEE international symposium on requirements engineering, pp. 249–262. IEEE (2001)
47. Westfechtel, B.: Case-based exploration of bidirectional transformations in qvt relations. Software & Systems Modeling **17**(3), 989–1029 (2018)
48. Yu, E.: Modeling strategic relationships for process reengineering. Social Modeling for Requirements Engineering **11**(2011), 66–87 (2011)
49. Yu, E.S.: An organization modeling framework for information system requirements engineering. In: Proc. Wkshp. Information Technologies and Systems (WITS'93), p. 9 (1993)
50. van Zee, M., Bex, F., Ghanavati, S.: Rationalization of goal models in GRL using formal argumentation. In: Proc. RE'15, pp. 220–225. IEEE Computer Society (2015)
51. van Zee, M., Bex, F., Ghanavati, S.: Rationalgrl: A framework for argumentation and goal modeling. Argument & Computation (Preprint), 1–55 (2020)
52. van Zee, M., Marosin, D., Bex, F., Ghanavati, S.: RationalGRL: A framework for rationalizing goal models using argument diagrams. In: Proc. ER'16, pp. 553–560. Springer (2016)
53. Zowghi, D., Offen, R.: A logical framework for modeling and reasoning about the evolution of requirements. pp. 247–257 (1997)