An Event-B Capability-Centric Model for Cloud Service Discovery

Aicha Sid'Elmostaphe^1, J Paul Gibson^2, Imen Jerbi^3, Walid Gaaloul^4, Mohamedade Farouk Nanne^5
Telecom SudParis, SAMOVAR, Institut Polytechnique De Paris, Paris, France^{1,2,4}
Faculty of Science and Technology-CSIDS, University of Nouakchott, Nouakchott, Mauritania^{1,5}
BYO Networks, Paris, France^3

***Abstract*—Cloud computing has become increasingly adopted due to its ability to provide on-demand access to computing resources. However, the proliferation of cloud service offerings has introduced significant challenges in service discovery. Existing cloud service discovery approaches are often evaluated solely through simulation or experimentation and typically rely on unstructured service descriptions, which limits their precision and scalability. In this work, we address these limitations by proposing a formally verified architecture for capability-centric cloud service discovery, grounded in the Event-B method. The architecture is built upon a capability-centric service description model that captures service semantics through property-value representations. A core element of this model is the formally verified variantOf relation, which defines specialization among services. We prove that variantOf satisfies the properties of a partial order, enabling services to be structured as a Directed Acyclic Graph (DAG) and thus supporting hierarchical and scalable discovery. We formally verify the consistency of our model across multiple refinement levels. All proof obligations generated by the Rodin platform were successfully discharged. A scenario-based validation further confirms the correctness of dynamic operations within the system.**

Keywords*—*Formal verification; cloud service discovery; capability modelling

I. Introduction

Cloud computing has emerged as a paradigm that changes the way IT services are delivered [1]. By proposing a multitude of on-demand services, cloud computing has become indispensable for companies seeking efficient workload management and cost-effective high-quality service [2]. The rapid growth in the number of cloud services has intensified the challenges of efficient service discovery [3]. Cloud service discovery aims to assist cloud users in locating the most suitable cloud services for their needs [4].

Although numerous approaches for identifying cloud services have been proposed including [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], current cloud service discovery solutions lack a structured and detailed definition of cloud services [17]. Furthermore, they fail to consider the continuous growth in the number of service offerings and the increasing heterogeneity of cloud services.

In addition to the lack of structure in service descriptions, the dynamic nature of the cloud environment, where services may frequently appear while others may disappear [3], adds further complexity to the discovery process. The cloud service discovery process involves both the description of services and a matchmaking mechanism that compares available offerings with user requirements. However, the unstructured nature of service descriptions increases the difficulty of performing accurate matchmaking, which in turn amplifies the overall complexity of the discovery process. These challenges underscore the need for approaches that can guarantee the correct behavior of cloud service discovery systems.

Despite recent progress, to the best of our knowledge, current cloud service discovery approaches have been exclusively evaluated through simulation and experiments. However, simulation-based approaches have not proven to be efficient in evaluating complex systems, particularly in terms of assessing their functional properties, including system correctness [18]. Similarly, while experiments provide valuable insight into performance metrics, they fall short in comprehensively assessing all possible system states or interactions.

In contrast, formal verification has shown promising results in ensuring the correctness of complex information systems. Formal verification allows us to verify the functional properties of complex and large-scale systems and specify the relationships between behavioral interactions within such systems. Formal verification can demonstrate the precision and correctness of these systems, making it an essential tool in the context of cloud computing [18]. Furthermore, formal verification offers a viable solution for addressing fundamental challenges in cloud service discovery [17], including reliability, scalability, and security, making it highly relevant for the advancement of this field.

Nevertheless, formal verification has received significantly less attention in the context of cloud service discovery compared to adjacent fields such as service composition [19], [20], [21], [22], [23], [24], [25]. To date, only a limited number of works [26], [27], [24] have tackled the formal verification of cloud service discovery systems.

While these efforts represent valuable contributions, they do not comprehensively address the full range of challenges inherent in cloud service discovery. In particular, to the best of our knowledge, no existing work formally verifies cloud service discovery systems that support multiple types of services and dynamic behavior.

To address these limitations, this paper proposes a new verification approach[1] that covers multiple aspects of cloud service discovery. Our approach ensures that (i) both services offered by cloud providers and requested by users are described

[1]For detailed proofs and the complete formal model, see: https://github.com/Aichasdm/Capability-centric-cloud-service-discovery-model.git

formally; (ii) the returned services satisfy required functional levels; and (iii) dynamic changes are handled without compromising the correctness of discovery operations. Given the complexity of verifying these requirements, we adopt the formalism of Event-B.

Event-B is a formal method for modeling and developing complex systems. Its objective is to build correct systems by construction through a series of refinements from abstract specifications to concrete implementations. Each refinement step is validated using mathematical proof obligations based on predicate calculus and typed set theory [28]. In this work, we formally verify a cloud service discovery architecture. Recognizing the benefits of using a service repository in the enhancement of the efficacy of cloud service discovery [4], along with the efficiency of tree-like structures, we propose an architecture that combines both elements for storing and organizing services.

Furthermore, we argue that the unstructured nature of cloud service descriptions is a major factor contributing to the complexity of service discovery. To address this, we adopt the Capability Model [29], a structured representation that not only mitigates heterogeneity and unstructured data but also supports highly configurable and dynamic service offers such as cloud offerings. The Capability Model enables the representation of both functional and non-functional service properties and provides a partial order between services, which can be structured as a directed acyclic graph.

The novelty of this work is twofold. First, it bridges the gap between cloud service discovery and formal verification, resulting in more reliable and consistent discovery systems. Second, it provides a formal verification of service description based on a generic and extensible model—the Capability Model. Our verification using the Event-B method offers a rigorous foundation that ensures the consistency and correctness of the model itself. The Capability Model has previously been applied in the context of Web services [30], business processes [31] and Network as a Service(NaaS) [32]. This verification supports its safe reuse across diverse service-oriented domains, including but not limited to cloud computing.

To summarize, the main contributions of this paper are:

- Verification of the cloud service description model;
- Verification of the consistency of the partial order between services;
- Verification of the behavior of the cloud service discovery process.

The remainder of this paper is as follows. Section II gives a review of related works; Section III provides our motivation and an overview of the proposed architecture; Section IV describes the formalization and verification of the consistency of the cloud description model and the partial order relation between services; Section V presents the formal model of the proposed cloud service discovery; Section VI discusses our findings. Finally, Section VII presents a conclusion.

II. Related Works

This section reviews existing approaches for cloud service discovery. In light of the numerous solutions proposed in recent years, we focus on the most frequently cited and contextually relevant works. Based on our research context, the approaches are classified into two categories. The first includes methods developed to formalize or verify cloud service discovery through mathematical or logical techniques. The second encompasses approaches validated through experimental or simulation-based evaluations only.

A. Formal Cloud Service Discovery Approaches

The following is a brief overview of research efforts that utilize formal methods to verify cloud service discovery systems. Notably, only a limited number of works in the literature directly address this problem.

In study [27], the authors propose a resource discovery model for grid computing based on a hierarchical tree structure, supporting multi-attribute queries. The model is verified using model checking techniques. The system behavior is decomposed into three components: data gathering, control, and discovery. This modular design facilitates maintenance, development, and verification. The relationships between these components are formalized using Binary Decision Diagrams (BDDs). Properties of the resource discovery process are specified using temporal logic (CTL and LTL) and verified to ensure they are satisfied. The authors claim the model demonstrates soundness, completeness, and consistency.

In study [26], a method is presented for discovering human resources in the Expert Cloud, with an emphasis on trust-based expert search. The system is modeled as an undirected, weighted graph where nodes represent human resources and edges indicate prior interactions. Each node is annotated with information relevant to the resource. The structural and compositional aspects of the discovery process are verified using the NuSMV[2] model checker, and the properties are defined in temporal logic. The authors argue that their model is sound, reachable, complete, deadlock-free, and consistent. However, the approach does not address key concerns such as quality of service (QoS), billing, or authorization.

Authors in study [24] aim to leverage the gap between formal verification and cloud service discovery and composition by proposing an architecture for formal service matching where behaviors are seen as ordered sequences of services. The proposed architecture allows the formal matching and composition of ordered sequences of services. The approach is based on Cloudle[8] and the ABCS framework[33].

While valuable, these approaches exhibit several limitations. First, they address only specific aspects of service discovery, such as trust in a particular domain, as in study [26], or consistency of ordered sequence of services [24]. Moreover, although the model proposed in study [27] is useful, it has been applied in the context of grid computing and has not been evaluated for cloud services.

[2]See https://nusmv.fbk.eu

B. Informal Cloud Service Discovery Approaches

In contrast to formal methods, a large number of approaches rely on simulations or experiments. These can be broadly categorized into ontology-based, keyword-based approaches, and hybrid approaches (ontology-based and keyword-based) following the classification in study [4]. This discussion is not intended as an exhaustive review of all existing approaches. Instead, it presents a representative selection of notable works to illustrate the key characteristics and trends within this category of cloud service discovery methods.

1) Ontology-based approaches: In study [6], Modica and Tomarchio propose a semantic discovery framework that facilitates the alignment between user demands and provider offerings by considering their respective utility and business objectives. The model incorporates seven ontologies, including shared concepts between user and provider perspectives (Support, mOSAIC, Application, SLA ontologies), provider-specific ontologies (Market and Offer), and a user-specific ontology (Request). To enable comparison between the user and provider perspectives, a set of mapping rules is defined. Furthermore, a semantic matchmaking process is integrated into the framework to evaluate the degree of similarity between user requests and provider offers, leveraging a semantic similarity algorithm.

Within a series of works [8], [34], [35], [36], Kang and Sim have proposed the Cloudle engine for discovering cloud services. It represents a cloud service search engine that consults a cloud ontology to reason about relationships among cloud services. The proposed architecture includes a query processor, a similarity reasoning utility based on a cloud ontology, and a price and timeslot utility. The query processor handles user queries and sends them to both the similarity reasoning utility and the price and timeslot utility agent. The former includes three similarity reasoning methods: (i) concept similarity reasoning, (ii) object property reasoning, and (iii) datatype property reasoning.

The engine was later extended, in study [37] with an agent-based testbed for cloud service discovery. The system architecture is based on multiple broker agents and trading agents (providers and users), with various applications and resources. The service discovery process involves four stages: selection, evaluation, filtering, and recommendation. Matching between user requests and provider specifications is performed using a cloud ontology, based on three similarity reasoning methods: concept similarity, property similarity, and datatype similarity.

In a further enhancement, the study in [38] presents a revised architecture for Cloudle. The main difference compared to the previous version is the introduction of crawlers. Crawlers are responsible for maintaining and updating the service database. They are deployed to collect information about cloud service providers from webpages, thereby keeping the Cloudle database up to date. The service reasoning process includes three reasoning types: similarity reasoning, compatibility reasoning, and numerical reasoning.

Finally, the study [39] proposes CB-Cloudle, an enhancement of Cloudle that introduces a centroid-based cloud search engine. It uses a dedicated crawler for each cloud provider and ranks cloud services based on the k-means clustering algorithm.

The research [40] introduces an ontology-based cloud service search engine called CSSE. The CSSE framework consists of three layers: the cloud service ontology layer, the cloud service identification layer, and the search engine user layer. The user layer provides a web interface that allows users to request cloud services by entering search keywords. The ontology layer includes a repository of cloud concepts generated by the cloud ontology builder, which is based on the NIST cloud computing standards and real-world cloud service metadata gathered in a prior work [41]. The identification layer contains a cloud service repository and a service identifier that detects potential cloud services through similarity checks between user queries and ontology concepts.

In study [42], the authors propose a platform for cloud service discovery. First, natural language processing (NLP) techniques are applied to automatically annotate cloud service descriptions with semantic content. Based on these annotations, each service is represented as a semantic vector. The platform enables semantic matching between user queries—written in natural language—and suitable cloud services.

The study [14] proposes a decentralized, peer-to-peer (P2P) semantic service discovery approach. A multi-layered cloud ontology is employed to represent service descriptions in a standardized and meaningful way. These service descriptions are stored in decentralized registries that collectively form the P2P network. Peers are grouped into clusters based on the semantic similarity of their service descriptions. Furthermore, Semantic Overlay Networks (SONs) are used to establish semantic links between peers, thereby enhancing the effectiveness of the discovery process.

To address fuzziness in user preferences, the study [10] proposes Cloud-FuSeR, a fuzzy, user-oriented cloud service selection system. It comprises: a fuzzy cloud ontology for computing similarity between user needs and services, a fuzzy Analytic Hierarchy Process (AHP) for deriving weights of non-functional properties, and a fuzzy TOPSIS method to rank services using AHP-derived weights and service performance.

2) Keyword-based approaches: The study [5] proposes an automated version of CSSE that extracts cloud service descriptions from the Web. Extracted features are clustered by similarity. The Service Detection and Tracking (SDT) model is introduced to support modeling and tracking services across providers.

In study [7], a two-stage recommendation model is presented. First, it analyzes unstructured textual descriptions of cloud services using Hierarchical Dirichlet Processes (HDP) to form clusters. Then, a Personalized PageRank algorithm ranks services based on tags, enabling personalized recommendations.

The study [15] proposes a cloud service recommendation system named CSRecommender, which enables users to search for cloud services and receive a list of relevant results based on queries and ratings. The system consists of five main components: a crawler, which collects potential cloud service descriptions from the web; a cloud service identifier, which verifies whether a webpage represents a valid cloud service; an indexer, which stores identified services in a structured repository; a search engine, which allows keyword-based queries; and a recommender system, which offers both collaborative and content-based recommendations.

In study [43] Focused Crawler for the Cloud service Discovery (FC4CD) is presented. This tool can identify, gather and analyze cloud services available on the Web. In study [44], the authors present a cloud service registry and discovery system designed to classify and identify services based on their underlying model (IaaS, PaaS, SaaS) and associated QoS attributes. Initially, cloud services are categorized into datasets by a service model. A decision tree-based identification algorithm then classifies user requests according to the relevant service model using QoS preferences provided by the cloud consumer. Once the model is identified, the Split and Cache (SAC) algorithm performs service discovery by retrieving cloud service providers (CSPs) whose offerings match the specified QoS requirements.

3) Hybrid approaches: In study [10], The authors propose a cloud service recommendation system based on semantic technologies, employing a fuzzy service ontology structure. Cloud service descriptions are parsed using natural language processing (NLP) techniques to identify and extract key concepts, which are subsequently used to populate a fuzzy ontology for cloud services. User queries, also expressed in natural language, are parsed to extract fuzzy connectives (e.g., AND, OR) and are represented as logical expressions. These expressions are then translated into first-order Horn clause logic and further refined using disjunctive normal form (DNF) transformations to generate multiple query candidates. The matching process is carried out by evaluating semantic similarity between user requirements and ontology concepts, utilizing SPARQL queries to retrieve relevant cloud services.

In study [12], the authors present an LDA-based Self-Adaptive Semantic Focused (LDA-SSF) crawler designed to efficiently collect, categorize, and store Cloud services by leveraging a Cloud Service Ontology (CSOnt) to calculate semantic similarity. To enhance the crawling process, a URLs Priority technique, based on Term Frequency–Inverse Document Frequency(TF-IDF) and semantic similarity, is employed to assign priority scores to candidate URLs by computing their textual similarity to a given Cloud service category. Moreover, an ontology-learning technique, based on the LDA model and semantic distance, is proposed to automatically enrich the CSOnt with new concepts, thereby maintaining the crawler's performance over time.

All the approaches presented in this section have been evaluated only through simulation or experimental validation. As previously stated, such evaluations cannot guarantee the correctness of the discovery system or identify hidden flaws. Furthermore, these approaches lack standardization in cloud service discovery. Furthermore, while formal methods have been applied to various aspects of cloud computing, such as service composition and orchestration [19], [20], [21], [22], [23], [24], [25], these works typically address services already deployed within cloud environments. In contrast, our work focuses on the discovery of services offered by cloud providers. This phase has received limited attention in formal specification literature. We formalize this process using Event-B, capturing both the static structure of capability-based descriptions and the dynamic behavior of user-driven discovery over an evolving service repository. To our knowledge, this constitutes the first refinement-based Event-B formalization dedicated specifically to cloud service discovery.

III. Approach Overview and Motivation

Cloud service discovery systems assist users in identifying suitable services offered by various providers. These systems must operate in highly dynamic environments, where providers frequently update, add, or remove their service offerings. Users typically send requests specifying both functional and non-functional requirements such as service type, region, pricing, and specific capabilities. The system is expected to return services that fully satisfy the given constraints.

However, unstructured service descriptions and ambiguous user inputs can lead to significant mismatches between requested and returned services. In addition, the evolving nature of cloud offerings requires that discovery systems handle service updates while ensuring continued correctness and consistency. Without formal verification, such systems may have subtle faults that remain undetected during simulation or empirical testing [27]. To address these challenges, this work proposes a repository-based cloud service discovery system whose behavior and correctness are formally verified using the Event-B method. At the core of this architecture is the Capability Model [29], [45], which provides a structured representation of service properties and supports abstraction, variability, and semantic alignment between services and user needs.

Fig. 1 illustrates the proposed architecture. Initially, cloud users submit service requests described using the Capability Model. These requests are first validated for consistency by the request/offer checker component. Once validated, the request is forwarded to the matching engine, responsible of the matchmaking process, which consults the repository of cloud service offerings to identify suitable candidates. The resulting list is then returned to the user.

The adopted matchmaking strategy is deliberately simple and deterministic. We do not currently support partial satisfaction or ranking of offers; only services that fully satisfy the constraints of the request (or generalize it) are returned. While advanced discovery techniques, such as similarity-based ranking or graph traversal over variant hierarchies, are common in practical systems, they are out of scope in the present work. Our focus is on verifying the logical foundations of cloud service discovery behavior and ensuring correctness through refinement and theorem proving.

The Capability Model enables expressive yet semantically rigorous service descriptions, capturing both functional aspects (e.g., service type) and non-functional properties (e.g., price, performance). Unlike traditional approaches such as WSDL, it supports semantic alignment and abstraction through relations such as `specifies`, `extends`, and `variantOf`, which facilitate service categorization and generalization.

Although details about these inter-capability relations and their formal properties are introduced in the next section, it is important to note that they enable the repository to be managed as a graph structure that supports reasoning about

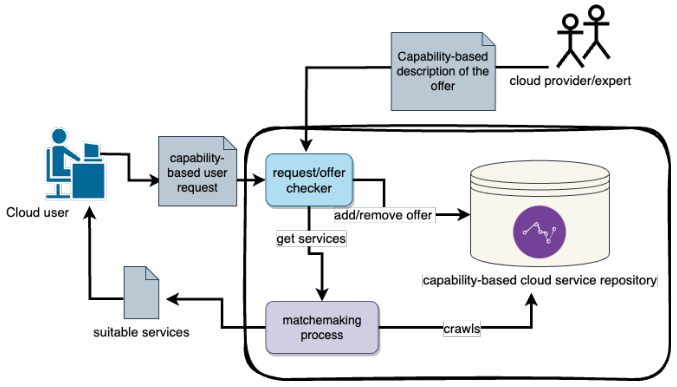


Fig. 1. Overview of the adopted architecture.

compatibility and substitution between services. These features lay the foundation for scalable and correct service discovery under evolving system configurations.

The next section presents the formalization and verification of this architecture. We begin by specifying the structural properties of the Capability Model and proceed to formalize the discovery behavior under dynamic conditions using refinement in Event-B.

IV. Formal Modeling and Verification of the Capability Model

In this section, we present our formal model for service description. Since our model is developed using the Event-B method, we begin with a brief overview of its core concepts. We then introduce the Capability Model [29], a capability-centric service description model. Following this, we present the Event-B formalization of the model in detail, and conclude with its validation using a motivating example. As noted in study [29], the Capability Model is sufficiently expressive to describe a wide range of service-oriented systems, including cloud services, web services, and business processes. Accordingly, this work contributes to the formal validation of the Capability Model by ensuring its correctness under rigorously defined structural and behavioral conditions.

A. The Event-B Method

Event-B [28] is a formal method for modeling and developing complex systems through a correctness-by-construction approach. It supports system development via a series of refinement steps, transitioning from an abstract specification to a more concrete and implementable model. Each refinement step must preserve correctness and is validated through the generation and discharge of proof obligations.

An Event-B model consists of two main components: *contexts* and *machines*. The context defines the static part of the model, including sets, constants, axioms, and theorems. In contrast, the machine captures the dynamic behavior of the system, and includes variables, invariants, and events. Events represent atomic transitions; each is defined by parameters, guards (which control when an event may occur), and actions (which update the state variables).

Correctness is ensured by discharging proof obligations that validate invariant preservation, guard strengthening, well-definedness, and other formal properties. These obligations are automatically or interactively verified using the Rodin platform [46], which integrates theorem provers for Event-B. In this work, Event-B is used to model and verify a capability-centric cloud service discovery system. We structure the model across multiple refinement levels, using formal proofs to guarantee that both structural and behavioral properties are preserved throughout development.

B. Overview of the Capability Model

Service description plays a fundamental role in the automation of service discovery and in achieving interoperability in web-based environments [47]. The concept of capability has emerged as a central element in service description [48]. To enable automated discovery, service descriptions must explicitly define the capabilities of services, thereby allowing users to identify and select services based on their functionality rather than relying on informal documentation to infer what a service can perform [47].

The Capability Model was initially proposed in study [29], and a subsequent formalization was presented in study [45]. In this paper, we focus on the version introduced in study [45], providing a complete and rigorous formalization using the Event-B method. Where appropriate, we refer back to the original model in study [29] to ensure alignment with its conceptual foundations. To validate the coherence of our formalization with the original model, we reuse the motivating example presented in study [45]. This allows us to demonstrate that our Event-B specification conforms to the same structural and behavioral assumptions. We have made minor corrections to the original example where discrepancies were observed, thereby illustrating the added value of formal verification and theorem proving in identifying subtle inconsistencies.

1) The model components: The core concept in the Capability Model [29] is that of a *capability*, which defines the functionality a service can provide. The model describes a service in terms of its capability, represented as a set of *property entries* (or attributes). Each property entry consists of a pair $(Property, Value)$, where $Property$ denotes a service property (e.g., destination, price), and $Value$ represents the set of possible values associated with that property. Both property and value refer to ontological terms.

A $Property\ entry$ is defined with respect to a triplet $(Property, MGV, SR)$ where MGV (most general value) is the domain of values that the property can take, and SR is a specification relation that could be defined over elements of MGV with respect to the meaning of the property $Property$.

To illustrate this, consider the example shown in Fig. 2, in which a shipping company offers four service configurations to accommodate diverse customer types.

- C1: A standard offer that provides a basic shipping service to Australia without any constraints on package weight or destination.
- C2: A specialized offer for packages to Australia, restricted to those with a weight less than or equal to 100

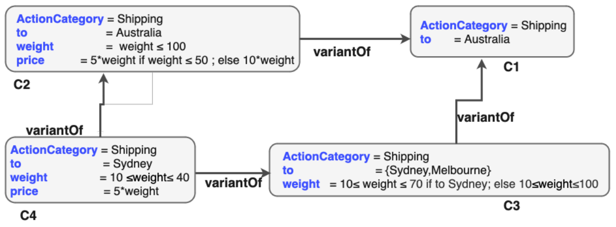

ActionCategory = Shipping
to = Australia
weight = weight ≤ 100
price = 5*weight if weight ≤ 50 ; else 10*weight
C2
variantOf
ActionCategory = Shipping
to = Australia
C1
variantOf
variantOf
ActionCategory = Shipping
to = Sydney
weight = 10 ≤weight≤ 40
price = 5*weight
C4
variantOf
ActionCategory = Shipping
to = {Sydney,Melbourne}
weight = 10≤ weight ≤ 70 if to Sydney; else 10≤weight≤100
C3

Fig. 2. A motivation example for the variantOf relation between capability from [45].

kg. The pricing is conditional: if the weight is less than or equal to 50 kg, the price is calculated as $5 \times$ weight; otherwise, it is $10 \times$ weight.

- C3: A further specialized offer for packages destined to Sydney and Melbourne. If the destination is Sydney, the acceptable weight range is between 10 and 70 kg; otherwise, the acceptable weight range is between 10 and 100 kg.
- C4: A specialized offer for packages to Sydney, restricted to those with a weight between 10 and 40 kg. the price is calculated as $5 \times$ weight

Thus, as illustrated in Fig. 2, capability $C2$ conforms to the Capability Model. For example, the property entry $to, Australia$ is defined with respect to the triplet $(\texttt{to}, \texttt{GeographicalLocation}, \texttt{locatedIn})$, where $\texttt{GeographicalLocation}$ denotes the most general value of *to*, and $\texttt{locatedIn}$ expresses a specification relation such that, for instance, $Sydney$ is considered to be $locatedIn$ $Australia$.

Furthermore, a specific property that is shared among all capabilities within the same domain is $actionCategory$. This property specifies the category of action that the capability can achieve. For instance, in Fig. 2, all capabilities have the value $shipping$ for the property $actionCategory$. In addition, the capability model allows for the definition of complex types of property values. In particular, there are five types of values for a property $SingleValue$, $ConstrainedValue$, $FunctionalValue$, $ConditionalValue$, and $EnumeratedValue$. $SingleValue$ refers to values of the type instance or subclass. For example, in Fig. 2 Sidney can be seen as an instance of Australia that is itself a subclass of $GeographicalLocation$. $ConstrainedValue$ allows to make a constraint on the value of a property. For example, in Fig. 2, the property $weight$ in $C2$ is of type $ConstrainedValue$. $FunctionalValue$ is used to define relationships between properties of the same capability. For example, the value of the property price in $C4$ is of type $FunctionalValue$. $ConditionalValue$ enables the definition of a value that depends on the value of another property. As an example, the value of the property $weight$ in $C2$ is of type $ConditionalValue$. Finally, $EnumerationValue$ denotes a finite set of a property values. For instance, the value of the property to within $C3$ is of type $EnumerationValue$.

It worth to note that, the semantics of the specification relation SR are domain-dependent and should, if possible, be defined with respect to the meaning of the values within each MGV. For instance, in numerical domains, for example, specification relation between elements may be simply the set inclusion (e.g., $\{10, 20\} \subseteq \{0..100\}$), while in geographical domains, it may be represented by a containment relation such as $locatedIn$. In networking domains, such as IP address hierarchies, specification could rely on subnet inclusion.

variantOf Relation: In addition to modeling relationships between the properties of a capability, the Capability Model introduces relations between capabilities to support hierarchical structuring. In study [45], the authors define such a hierarchy through the $variantOf$ relation, which captures when one capability is more specific than another. This relation generalizes the sub-relations $specifies$ and $extends$, and is defined based on an extension of the specification relation $\texttt{SR}$ to operate over sets of values within a shared MGV.

The $variantOf$ relation is formally defined as follows:

A capability C_1 is said to be $variantOf$ C_2 if the following two conditions hold:

- For every property p in C_1, the value assigned to p in C_1 is either equal to the value of p in the extended capability C_2/C_1, or it $specifies$ the value of p in C_2/C_1;
- There exists at least one property p in C_1 for which the value of p in C_1 strictly $specifies$ the corresponding value in C_2/C_1.

Here, C_2/C_1 denotes the extension of capability C_2 by C_1, i.e., a merged capability that includes all properties of C_2 and any additional properties from C_1.

The $\texttt{specifies}$ relation between two value sets v_1 and v_2 is defined as:

$$v_1 \ \texttt{specifies} \ v_2 \iff (v_1 \subset v_2 \lor \exists \underline{SR}.v_1 \mapsto v_2 \in \underline{SR})$$

where MGV is the most general value domain associated with the property and $\underline{SR}$ is a specification relation between twos sets of MGV.

C. The Event-B Model Architecture

In this section, we present our Event-B formalization of the Capability Model.

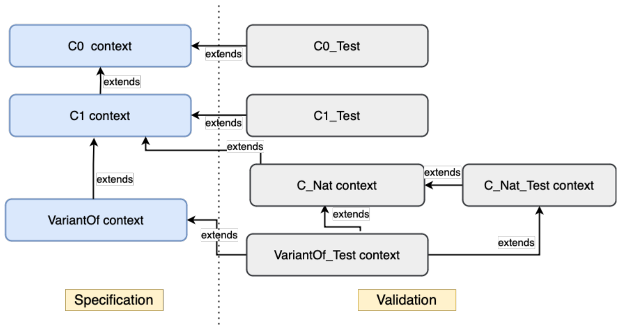


Fig. 3. Architecture of the Event-B model for the verification of the capability model.

The model is structured into three contexts (see Fig. 3):

- *C0* defines the core concepts of the Capability Model, including property value types;
- *C1* formalizes the specification relation between capabilities;
- *VariantOf* defines the variantOf relation and includes proofs that it constitutes a partial order, along with several inference theorems to deduce variantOf relations between capabilities.

Fig. 4 illustrates the corresponding Event-B model of the context C0. We define three foundational sets: i)PROPERTY: the set of domain-specific properties (e.g., actionCategory, to, weight, price in the shipping domain); ii) GENERALVALUES: the set of most general property values (MGVs), including values such as natural numbers, locations, and other domain-relevant categories; iii) Expression: a set of logical expressions used to define constraints over property values.

Other core elements such as Capability, PropertyEntry, and possibleValues are defined as constants within the model. The set possibleValues, a subset of GENERALVALUES, represents all possible values a property may take within a given capability. To capture semantic of different property values types, we define specific subsets of possibleValues (e.g ConstrainedValue, ConditionalValue,etc.). A PropertyEntry is defined as a pair from the cartesian product of PROPERTY and a subset of possibleValues. Additionally, The relation between a property and its MGV has been established by the relation hasMGV that maps each property in PROPERTY with its domain of valid values in GENERALVALUES. For example, hasMGV(to) = GeographicalLocation , hasMGV(weight) = $\mathbb{N}$, and hasMGV(price) = $\mathbb{N}$, with the assumptions that GeographicalLocation $\subseteq$ GENERALVALUES, $and \mathbb{N} \subseteq$ GENERALVALUES.

Given this, for instance, valid propertyEntry include (weight, $\{10\}$) and (weight, $\{20, 30, 40, 60\}$). A capability is then represented as a set of such property entries. For example, $C1$ in the motivating example can be modeled in our model as

$$\mathtt{C1} = \{\mathtt{actionCategory} \mapsto \mathtt{shipping},\ \mathtt{to} \mapsto \mathtt{Australia}\}$$

where $\{\mathtt{actionCategory}, \mathtt{to}\} \subseteq \mathtt{PROPERTY}$, shipping is the hasMGV(actionCategory), and Australia is a set that belongs to GeographicalLocation.

Property value Types: SingleValues. This refer to a subset of GENERALVALUES that represent either a singleton or a set of multiple elements. Given that, we are using Rodin which is based on set theory, we do not need to use this constant as it is already integrated in Event-B language as a subset of possibleValues.

ConstrainedValues. A ConstrainedValue is defined through a constructor that maps a property and a logical expression to a set of values that satisfy the expression within the property's most general value. This constructor is represented as: $\mathtt{Const} \in \mathtt{PROPERTY} \times \mathtt{Expression} \to \mathcal{P}(\mathtt{ConstrainedValues})$

The semantics of Const are given by the following definition: $\mathtt{Const}(p \mapsto \mathtt{exp}) = \{x \mid \mathtt{satisfies}(x, \mathtt{exp}) = \mathtt{TRUE} \wedge x \in \mathtt{hasMGV}(p)\}$. That is, $\mathtt{Const}(p \mapsto exp)$ returns the subset of hasMGV(p) whose elements satisfy the condition that the expression exp evaluates to TRUE. . This relies on the satisfies relation, which is defined as: $\mathtt{satisfies} \in \mathtt{GENERALVALUES} \times \mathtt{Expression} \to \mathtt{BOOL}$. As an example, in Fig. 2, the value of the property weight in C_2 can be represented using a constrained value:

```
Const(weight ↦ makeExpression(
        makeGeneralvaluesFromNats(
        allNatLessThanOrEqualNat(100))))
```

Here, $\mathtt{Const}(\mathtt{weight} \mapsto \mathtt{exp})$ defines a constraint on the property weight, where exp is the expression makeExpression(makeGeneralvaluesFromNats(allNatLessThanOrEqualNat(100))). This expression is satisfied by all natural numbers less than or equal to 100, i.e., satisfies(x, exp) evaluates to TRUE for every $x \in \mathbb{N}$ such that $x \leq 100$.

ConditionalValues. A ConditionalValue represents dependencies between the values of two properties within a capability. We define it via a constructor that maps a capability, a property, two sets of possible values, and a Boolean condition to a set of ConditionalValues. This constructor is defined as: $\mathtt{CND} \in \mathtt{Capability} \times \mathtt{PROPERTY} \times \mathcal{P}(\mathtt{GENERALVALUES}) \times \mathcal{P}(\mathtt{GENERALVALUES}) \times \mathtt{BOOL} \to \mathcal{P}(\mathtt{ConditionalValues})$

The semantics of CND are governed by the relation: $\mathtt{condition} \in \mathtt{GENERALVALUES} \times \mathtt{Expression} \to \mathtt{BOOL}$. Given a capability cap, a property p, and a condition dependent on another property p_c, the meaning of a conditional value is as follows:

$$\mathtt{CND}(\mathtt{cap} \mapsto \mathtt{price} \mapsto \mathtt{v\_p} \mapsto \mathtt{v\_p2} \mapsto \mathtt{condition}(\mathtt{cap} \mapsto \mathtt{p\_c} \mapsto \mathtt{v\_c})) =$$

$$\begin{cases} \mathtt{v\_p} & \text{if } \mathtt{condition}(\mathtt{cap} \mapsto \mathtt{p\_c} \mapsto \mathtt{v\_c}) = \mathtt{TRUE} \\ \mathtt{v\_p2} & \text{otherwise} \end{cases}$$

Here, p is the property being assigned a conditional value, and p_c is the property whose value influences the selection. v_p and v_p2 are sets of values from the MGV of p , and v_c is a set of values from the MGV of p_c.

Example. In our motivating example (Fig. 2), the value of price in capability C_2 is of type ConditionalValue. It can be written as:

$$\begin{aligned}&\mathtt{CND}(\mathtt{C2} \mapsto \mathtt{price} \mapsto\\ &\quad \mathtt{FN}(\mathtt{C2} \mapsto \mathtt{price} \mapsto \mathtt{weight} \mapsto \mathtt{f5}) \mapsto\\ &\quad \mathtt{FN}(\mathtt{C2} \mapsto \mathtt{price} \mapsto \mathtt{weight} \mapsto \mathtt{f10}) \mapsto\\ &\quad \mathtt{condition}(\mathtt{C2} \mapsto \mathtt{weight} \mapsto\\ &\quad \mathtt{Const}(\mathtt{weight} \mapsto \mathtt{makeExpression}(\\ &\quad \mathtt{makeGeneralvaluesFromNats}(\\ &\quad\quad \mathtt{allNatLessThanOrEqualNat}(50)))))\end{aligned}$$

This expression states that if the weight is less than or equal to 50, then the price is computed using function f_5(that returns $5 \times \mathtt{weight}$); otherwise, it is computed using f_{10}(that returns $10 \times \mathtt{weight}$).

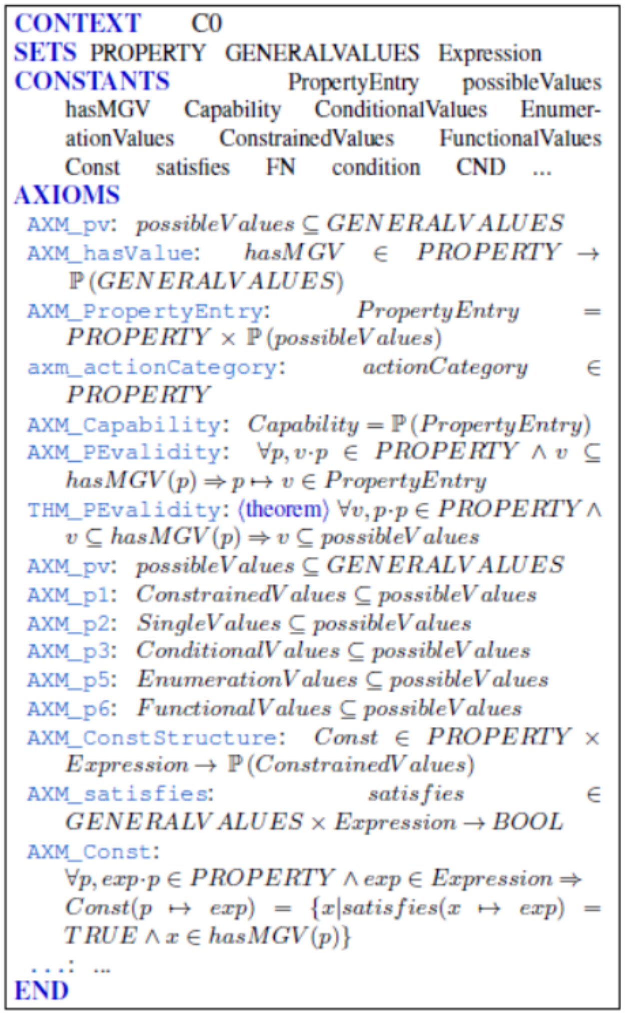
FunctionalValues. A `FunctionalValue` is defined by a constructor that maps a capability, a target property, a source property, and a function to a set of `FunctionalValues`. The constructor is given by: $\mathtt{FN} \in \mathtt{Capability} \times \mathtt{PROPERTY} \times \mathtt{PROPERTY} \times \mathtt{f} \to \mathcal{P}(\mathtt{FunctionalValues})$ where `f` is the set of partial functions defined as: $\mathtt{f} = \mathtt{PROPERTY} \times \mathtt{GENERALVALUES} \rightharpoonup \mathtt{GENERALVALUES}$. The semantics of a `FunctionalValue` define how the value of one property is computed based on the value of another property. Formally, the interpretation of: $\mathtt{FN}(\mathtt{cap} \mapsto \mathtt{p} \mapsto \mathtt{pf} \mapsto \mathtt{fn})$ is the set: $\{\mathtt{fn}(\mathtt{pf} \mapsto \mathtt{x}) \mid \mathtt{x} \in \mathtt{getPossibleValuesOfPonCapability}(\mathtt{cap} \mapsto \mathtt{pf}) \wedge (\mathtt{pf} \mapsto \mathtt{x}) \in \mathtt{dom}(\mathtt{fn})\}$ `cap` is a capability, `p` is the property within `cap` whose value is determined by a `FunctionalValue`, `pf` is another property within `cap` whose value is used as input to the function `fn`, $\mathtt{fn} \in \mathtt{f}$ is a partial function in the most general value domain of `p`, and $\mathtt{getPossibleValuesOfPonCapability}(\mathtt{cap}, \mathtt{pf})$ returns the possible values of property `pf` within capability `cap`.

Example. In our motivating example, the value of `price` within capability C_4 is defined as a `FunctionalValue`: $\mathtt{FN}(c_4 \mapsto \mathtt{price} \mapsto \mathtt{weight} \mapsto f_5)$. The function f_5 computes the price as five times the weight. It is defined as: $f_5 \in \mathtt{PROPERTY} \times \mathbb{N} \to \mathbb{N}$ with semantics given by: $f_5(\mathtt{weight} \mapsto ng) = \mathtt{makeGeneralvalueFromNat}(5 \times \mathtt{getNatForGeneralValue}(ng))$. This means that the price is computed as $5 \times \mathtt{weight}$, using the numeric interpretation of the general value. This expresses that for every value x of the source property pf that exists in the current capability and lies in the domain of function fn, the function computes the corresponding value of property p as $fn(pf \mapsto x)$.

Capability Validity. To ensure the well-structured definition of services, we define the notion of a *valid capability*. A capability is considered valid if it satisfies the following conditions:

(i) It includes a property denoted as `actionCategory`, representing the action performed by the capability;
(ii) It does not contain duplicate properties with conflicting values;
(iii) All property values lie within the domain defined by their associated `MGV`.

This constraint is defined as an axiom at the context level, by the constant `Capability_valid`. During any interaction with the system, we check whether a given capability belongs to the set of valid capabilities.



```
CONTEXT    C0
SETS PROPERTY  GENERALVALUES  Expression
CONSTANTS          PropertyEntry    possibleValues
    hasMGV   Capability   ConditionalValues   Enumer-
    ationValues     ConstrainedValues     FunctionalValues
    Const    satisfies    FN    condition    CND   ...
AXIOMS
 AXM_pv:  possibleValues ⊆ GENERALVALUES
 AXM_hasValue:   hasMGV  ∈  PROPERTY →
    ℙ(GENERALVALUES)
 AXM_PropertyEntry:           PropertyEntry          =
    PROPERTY × ℙ(possibleValues)
 axm_actionCategory:          actionCategory         ∈
    PROPERTY
 AXM_Capability:  Capability = ℙ(PropertyEntry)
 AXM_PEvalidity:   ∀p, v·p ∈ PROPERTY ∧ v ⊆
    hasMGV(p) ⇒ p ↦ v ∈ PropertyEntry
 THM_PEvalidity: ⟨theorem⟩ ∀v, p·p ∈ PROPERTY ∧
    v ⊆ hasMGV(p) ⇒ v ⊆ possibleValues
 AXM_pv:  possibleValues ⊆ GENERALVALUES
 AXM_p1:  ConstrainedValues ⊆ possibleValues
 AXM_p2:  SingleValues ⊆ possibleValues
 AXM_p3:  ConditionalValues ⊆ possibleValues
 AXM_p5:  EnumerationValues ⊆ possibleValues
 AXM_p6:  FunctionalValues ⊆ possibleValues
 AXM_ConstStructure:   Const ∈ PROPERTY ×
    Expression → ℙ(ConstrainedValues)
 AXM_satisfies:                 satisfies            ∈
    GENERALVALUES × Expression → BOOL
 AXM_Const:
    ∀p, exp·p ∈ PROPERTY ∧ exp ∈ Expression ⇒
    Const(p ↦ exp) = {x|satisfies(x ↦ exp) =
    TRUE ∧ x ∈ hasMGV(p)}
 ...:  ...
END
```

Fig. 4. Snapshot of the Event-B context C0 defining the core components.

Specification Relation. The specification relations described above, denoted `SR` and $\underline{\mathtt{SR}}$, define, respectively, relationships between individual values and between sets of values within the same most general value (MGV) domain. These relations are formally defined in Event-B within context `C1`, a portion of which is shown in Fig. 5.

We distinguish between two levels of specification:

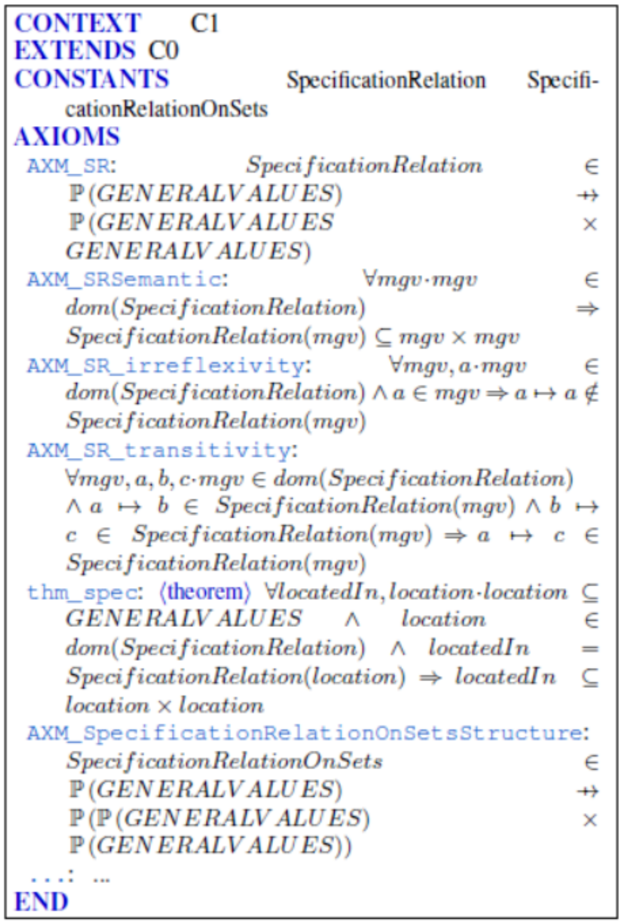
- SpecificationRelation (`SR`): a relation between individual values of a given MGV, defined as: $\mathtt{SpecificationRelation} \in \mathcal{P}(\mathtt{GENERALVALUES}) \rightharpoonup \mathcal{P}(\mathtt{GENERALVALUES} \times \mathtt{GENERALVALUES})$

- SpecificationRelationOnSets ($\underline{\mathtt{SR}}$): a relation between sets of values from a given MGV, defined as: $\mathtt{SpecificationRelationOnSets} \in \mathcal{P}(\mathtt{GENERALVALUES}) \rightharpoonup \mathcal{P}(\mathcal{P}(\mathtt{GENERALVALUES}) \times \mathcal{P}(\mathtt{GENERALVALUES}))$

Specifies Relation. We define the relation `specifies`, which holds between two capabilities that share the same property. It states that the value in one capability specializes the value in the other. Formally: $\texttt{specifies} \in \texttt{PROPERTY} \times \texttt{Capability\_valid} \times \mathcal{P}(\texttt{GENERALVALUES}) \leftrightarrow \texttt{Capability\_valid} \times \mathcal{P}(\texttt{GENERALVALUES})$. The semantics of this relation are given by:

$$(p \mapsto c_1 \mapsto v_1) \mapsto (c_2 \mapsto v_2) \in \texttt{specifies} \Leftrightarrow (v_1 \subset v_2 \vee (\texttt{hasMGV}(p) \in \texttt{dom}(\texttt{Specification RelationOnSets}) \wedge (v_1 \mapsto v_2) \in \texttt{Specification RelationOnSets}(\texttt{hasMGV}(p)))$$

Where p is a property shared by the capabilities c_1 and c_2, v_1 is the value of p in c_1, and v_2 is its value in c_2. Intuitively, the `specifies` relation holds either when there is a strict inclusion between v_1 and v_2, or when a domain-specific specification relation exists for the property's most general value, indicating that v_1 is more specific than v_2.



CONTEXT C1
EXTENDS C0
CONSTANTS SpecificationRelation SpecificationRelationOnSets
AXIOMS

AXM_SR: $SpecificationRelation \in \mathbb{P}(GENERALVALUES) \mathrel{+\!\!\!\rightarrow} \mathbb{P}(GENERALVALUES \times GENERALVALUES)$

AXM_SRSemantic: $\forall mgv \cdot mgv \in dom(SpecificationRelation) \Rightarrow SpecificationRelation(mgv) \subseteq mgv \times mgv$

AXM_SR_irreflexivity: $\forall mgv, a \cdot mgv \in dom(SpecificationRelation) \wedge a \in mgv \Rightarrow a \mapsto a \notin SpecificationRelation(mgv)$

AXM_SR_transitivity: $\forall mgv, a, b, c \cdot mgv \in dom(SpecificationRelation) \wedge a \mapsto b \in SpecificationRelation(mgv) \wedge b \mapsto c \in SpecificationRelation(mgv) \Rightarrow a \mapsto c \in SpecificationRelation(mgv)$

thm_spec: ⟨theorem⟩ $\forall locatedIn, location \cdot location \subseteq GENERALVALUES \wedge location \in dom(SpecificationRelation) \wedge locatedIn = SpecificationRelation(location) \Rightarrow locatedIn \subseteq location \times location$

AXM_SpecificationRelationOnSetsStructure: $SpecificationRelationOnSets \in \mathbb{P}(GENERALVALUES) \mathrel{+\!\!\!\rightarrow} \mathbb{P}(\mathbb{P}(GENERALVALUES) \times \mathbb{P}(GENERALVALUES))$

. . .: ...

END

Fig. 5. Snapshot of the Event-B context C1 defining the basic specification relations.

Example. In our motivating example, the value of the `weight` property in capability C_4 is more specific than this in C_2. Formally:

$$\texttt{weight} \mapsto C_4 \mapsto \texttt{getPossibleValuesOfPonCapability}(C_4, \texttt{weight}) \mapsto (C_2 \mapsto \texttt{getPossibleValuesOfPonCapability}(C_2 \mapsto \texttt{weight})) \in \texttt{specifies}$$

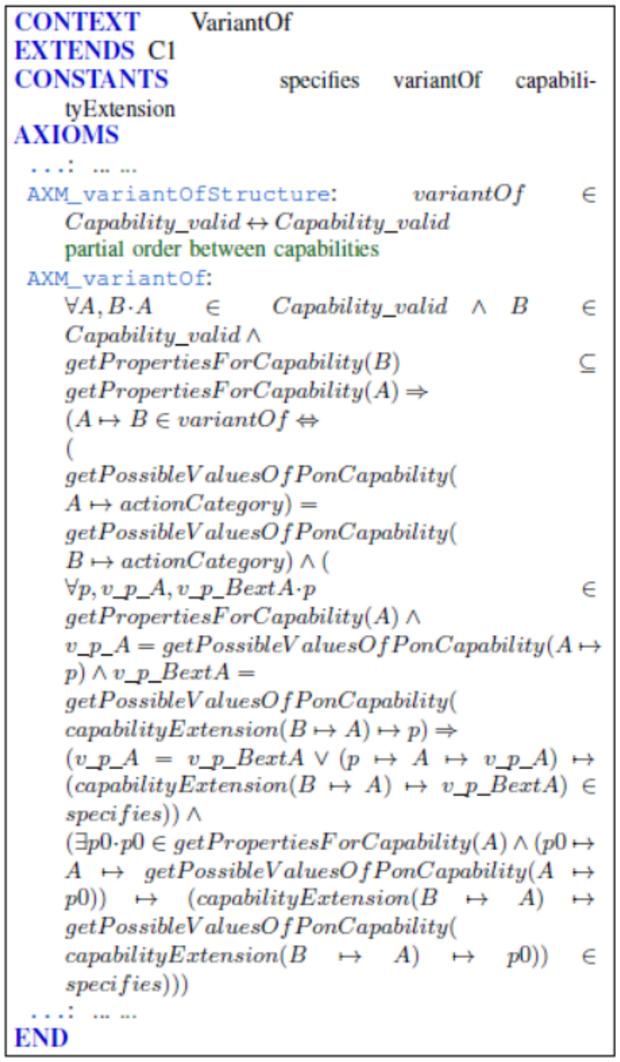
This holds because we have formally proved that: $\{10, \ldots, 40\} \subset \{0, \ldots, 100\}$.

VariantOf Relation. The model introduces a `variantOf` relation between capabilities. Its definition has been proposed in [45], and we have formally encoded this definition in Event-B. Fig. 6 illustrates the corresponding Event-B context. The definition of the `variantOf` relation is stated in axiom `AXM_variantOf`.

Additionally, we have formally proved that both `specifies` and `variantOf` are transitive and irreflexive. These proofs were conducted under the assumption that the relations SR and $\underline{SR}$ are themselves irreflexive and transitive, and that for any given `MGV`, it is not possible for both a strict inclusion and a $\underline{SR}$ relation to exist simultaneously between any two sets.

While in study [45], authors provide a rich conceptual and semi-formal foundation including inference rules and implementation support for discovering these relations, key correctness properties are only asserted but not formally verified. In contrast, our contribution offers a rigorous formalization of the model using the Event-B method. We encode the entire capability structure including property value types, the `specifies` and `variantOf` relations, and their supporting constraints into a provable specification. Moreover, our model introduces an additional consistency condition not enforced in [45]. Specifically, we require that every capability adheres to a well-formed structure (that we denote as a valid capability). This structural constraint enhances the reliability of capability definitions and ensure the soundness of subsequent reasoning. Therefore, we have defined these relations exclusively over valid capabilities, as reasoning about capability specialization is only meaningful when the capabilities themselves are structurally consistent.

Most importantly, we formally prove that the `variantOf` relation satisfies the properties of a partial order that were only affirmed informally in [45]. It is important to note that the partial order property of the `variantOf` relation directly enables the use of a directed acyclic graph (DAG) as the underlying structure of the service repository. By formally proving that `variantOf` is irreflexive and transitive, we ensure that the resulting capability graph does not contain cycles and maintains a coherent hierarchical structure. This acyclic and ordered structure is essential for efficient reasoning, enabling traversal operations (e.g., finding more generic or more specific capabilities) and supporting efficient service discovery [45]. Consequently, the correctness of the partial order lays the theoretical foundation for representing and managing cloud service variability using graph-based repositories.This insight provides a novel and formally grounded refinement of the original model. By discharging these proof obligations in Rodin, we reinforce the suitability of the Capability Model for correctness service discovery applications.



CONTEXT VariantOf
EXTENDS C1
CONSTANTS specifies variantOf capabilityExtension
AXIOMS
...: ... ...
AXM_variantOfStructure: $variantOf \in Capability\_valid \leftrightarrow Capability\_valid$
partial order between capabilities
AXM_variantOf:
$\forall A, B \cdot A \in Capability\_valid \land B \in Capability\_valid \land getPropertiesForCapability(B) \subseteq getPropertiesForCapability(A) \Rightarrow (A \mapsto B \in variantOf \Leftrightarrow (getPossibleValuesOfPonCapability(A \mapsto actionCategory) = getPossibleValuesOfPonCapability(B \mapsto actionCategory) \land (\forall p, v\_p\_A, v\_p\_BextA \cdot p \in getPropertiesForCapability(A) \land v\_p\_A = getPossibleValuesOfPonCapability(A \mapsto p) \land v\_p\_BextA = getPossibleValuesOfPonCapability(capabilityExtension(B \mapsto A) \mapsto p) \Rightarrow (v\_p\_A = v\_p\_BextA \lor (p \mapsto A \mapsto v\_p\_A) \mapsto (capabilityExtension(B \mapsto A) \mapsto v\_p\_BextA) \in specifies)) \land (\exists p0 \cdot p0 \in getPropertiesForCapability(A) \land (p0 \mapsto A \mapsto getPossibleValuesOfPonCapability(A \mapsto p0)) \mapsto (capabilityExtension(B \mapsto A) \mapsto getPossibleValuesOfPonCapability(capabilityExtension(B \mapsto A) \mapsto p0)) \in specifies)))$
...: ... ...
END

Fig. 6. Snapshot of the variantOf context.

D. Validation of the Model

We have validated the model incrementally by means of two complementary strategies.

First, we performed unit-level validation by constructing test contexts to verify the structural and semantic correctness of the individual model components. This approach is conceptually similar to unit testing in software engineering . The architecture of the resulting Event-b model is shown in Fig. 3. For each main context (e.g., $C0$), we created a corresponding test context (e.g., $C0\_Test$) in which key axioms were instantiated and verified as theorems. These theorems were then proven using the Rodin platform. This strategy allowed us to confirm that the model components were well-defined and consistent.

Second, we validated the model using a concrete, real-world scenario based on the motivating example described earlier. To support this, we introduced a new context, `C_Nat`, dedicated to the representation of natural numbers. This was necessary because Rodin does not allow multiple disjoint interpretations within a single abstract set such as `GENERALVALUES`. Within `C_Nat`, we defined natural numbers as a subset of `GENERALVALUES` and specified logical expressions such as `lessThan` and `greaterThan`. We also defined a mapping between abstract natural numbers and concrete numeric value.

The four capabilities presented in our motivating example were encoded in the `VariantOf_Test` context according to the Event-B model. We formally verified the validity of each capability and subsequently proved the `variantOf` relationships between them.

We argue that this validation strategy provides strong assurance of the model's soundness. All generated proof obligations were successfully discharged using the Rodin tool. The overall proof statistics are shown in Fig. 13.

V. Formal Modeling of Cloud Service Discovery

In this section, we present our Event-B model, which formalizes a cloud service discovery system based on the architecture shown in Fig. 1. We then verify and validate the model by discharging proof obligations and using a real-world scenario to demonstrate the correctness of the system's behavior.

A. The Event-B Model

Our Cloud Service Discovery (CSD) system is modeled as a formal Event-B refinement hierarchy built upon the previously defined Event-B Capability Model. This allows us to reason about discovery behavior while preserving the semantic correctness of service description. In this section, the terms *service*, *offer*, and *capability* refer to a cloud service offering described according to the Capability Model. The architecture of the final model is illustrated in Fig. 7.

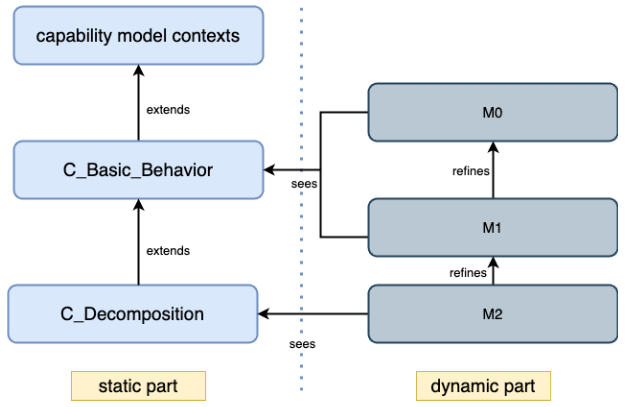
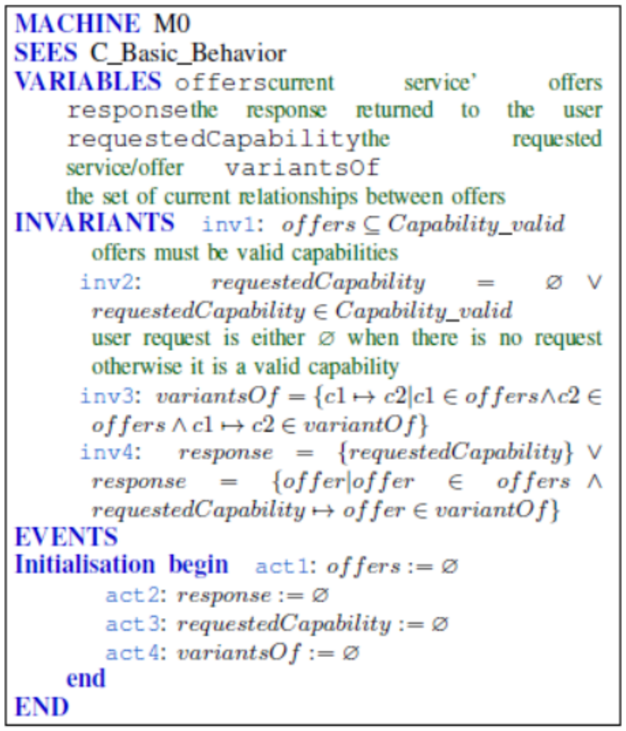


Fig. 7. The cloud service discovery formal model.

The system is modeled across three refinement levels, progressively transitioning from an abstract model to a more CSD concrete system.

Refinement Strategy. The formal development of the CSD model is organized as follows:

- M0 – Initial Model: This level captures the foundational behavior of the system. It is represented by the machine *M0*, which sees the context *C_Behavior* extending the capability model contexts and incorporating definitions of constants required to describe basic behavior. At this level, we model the elementary behavior of the CSD system. We begin by representing the repository as a graph, where `offers` denotes the set of current service offers (nodes), and `variantsOf` denotes the edges capturing the *variantOf* relation between offers. The system supports three primary interactions, formalized as events:

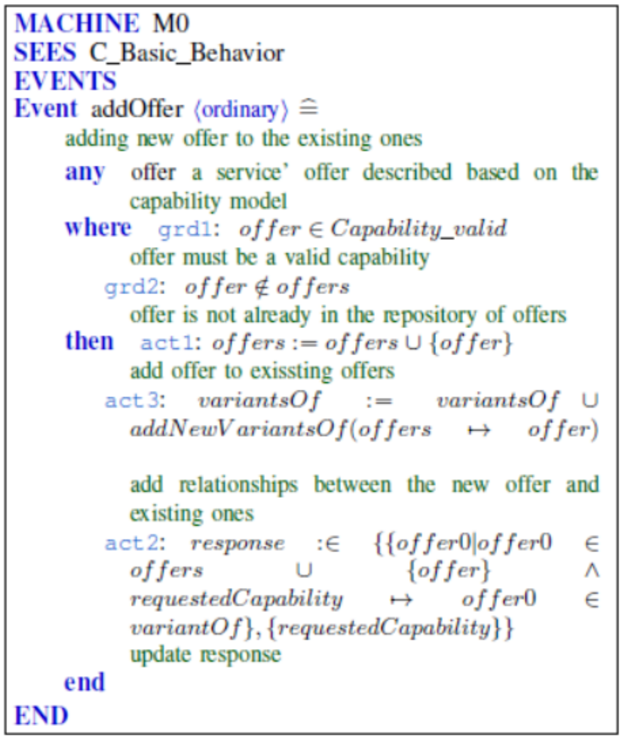


```
MACHINE M0
SEES C_Basic_Behavior
VARIABLES offerscurrent      service'      offers
    responsethe   response   returned   to   the   user
    requestedCapabilitythe               requested
    service/offer    variantsOf
    the set of current relationships between offers
INVARIANTS  inv1: offers ⊆ Capability_valid
      offers must be valid capabilities
    inv2:      requestedCapability    =    ∅   ∨
      requestedCapability ∈ Capability_valid
      user request is either ∅ when there is no request
      otherwise it is a valid capability
    inv3: variantsOf = {c1 ↦ c2|c1 ∈ offers∧c2 ∈
      offers ∧ c1 ↦ c2 ∈ variantOf}
    inv4:   response  =  {requestedCapability} ∨
      response   =   {offer|offer   ∈   offers  ∧
      requestedCapability ↦ offer ∈ variantOf}
EVENTS
Initialisation begin  act1: offers := ∅
        act2: response := ∅
        act3: requestedCapability := ∅
        act4: variantsOf := ∅
    end
END
```

Fig. 8. Snapshot of the Event-B M0 defining the initial model.

- ○ addOffers – introduces new service offers into the system;
- ○ removeOffers – removes existing service offers;
- ○ getOffersForNewRequest – initiates a discovery process based on a user requested service.

As illustrated in Fig. 8, the model also includes two additional variables: `requestedCapability`, representing the user's service request, and `response`, representing the potential returned services. The model remains abstract at this stage, as reflected by the non-deterministic invariants shown in Fig. 8. Nonetheless, it ensures that all services, whether requested or offered, are valid.
Initialization: At each refinement level, the system begins with an empty repository, no requests, and no response.
`addOffers` (Fig. 9): This abstract event inserts a valid offer (`Capability_valid`) into the repository. It updates the set of offers, adds edges to `variantsOf` using the `addNewVariantOf` relation, and modifies the `response` and `requestedCapability` accordingly.
`removeOffers`: Semantically similar to `addOffers`, this event removes a service offer and its associated edges from the repository and updates the response set.

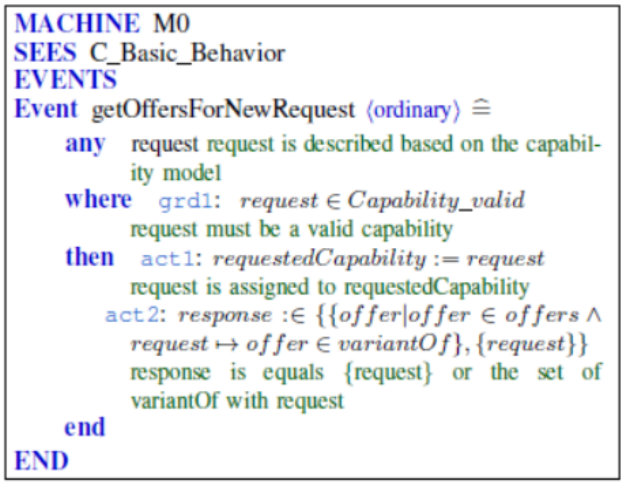


```
MACHINE M0
SEES C_Basic_Behavior
EVENTS
Event addOffer ⟨ordinary⟩ ≙
    adding new offer to the existing ones
    any   offer  a  service'  offer  described  based  on  the
          capability model
    where  grd1: offer ∈ Capability_valid
          offer must be a valid capability
        grd2: offer ∉ offers
          offer is not already in the repository of offers
    then   act1: offers := offers ∪ {offer}
          add offer to exissting offers
        act3:   variantsOf    :=    variantsOf  ∪
          addNewVariantsOf(offers    ↦    offer)

          add relationships between the new offer and
          existing ones
        act2: response  :∈  {{offer0|offer0   ∈
          offers      ∪      {offer}          ∧
          requestedCapability    ↦    offer0    ∈
          variantOf}, {requestedCapability}}
          update response
    end
END
```

Fig. 9. Formalization of the event addOffers.

`getOffersForNewRequest` (Fig. 10): This event models service discovery. Given a valid request, it updates `requestedCapability` and returns either an exact match or a set of more generic offers satisfying the `variantOf` relation.

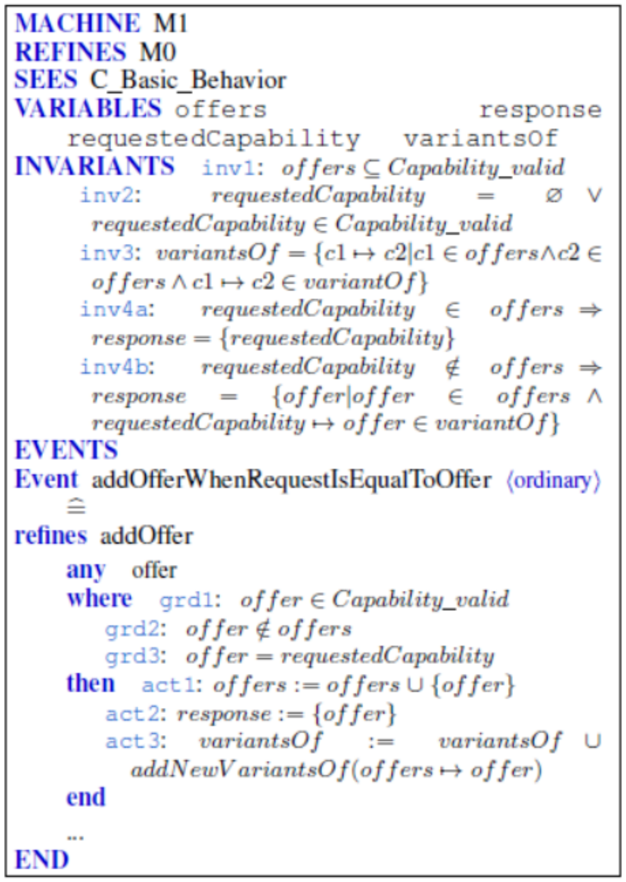


```
MACHINE M0
SEES C_Basic_Behavior
EVENTS
Event getOffersForNewRequest ⟨ordinary⟩ ≙
    any   request request is described based on the capabil-
          ity model
    where  grd1: request ∈ Capability_valid
          request must be a valid capability
    then   act1: requestedCapability := request
          request is assigned to requestedCapability
        act2: response :∈ {{offer|offer ∈ offers ∧
          request ↦ offer ∈ variantOf}, {request}}
          response  is  equals  {request}  or  the  set  of
          variantOf with request
    end
END
```

Fig. 10. Formalization of the event getOffersForNewRequest.

- M1 – First Refinement: (Fig. 11) This level, represented by machine *M1* (which also sees *C_Behavior*), refines the abstract model by introducing more concrete behaviors. In particular, it refines the three core events (`addOffers`, `removeOffers`, and `getOffersForNewRequest`) into specialized versions, and provides more precise invariants.

- addOfferWhenRequestIsEqualToOffer, addOfferWhenOfferIsVariantOfRequest, and addOffer: These events distinguish whether the user request matches an existing offer exactly, is a variantOf an offer, or is unrelated. Each case results in different updates to response and requestedCapability.

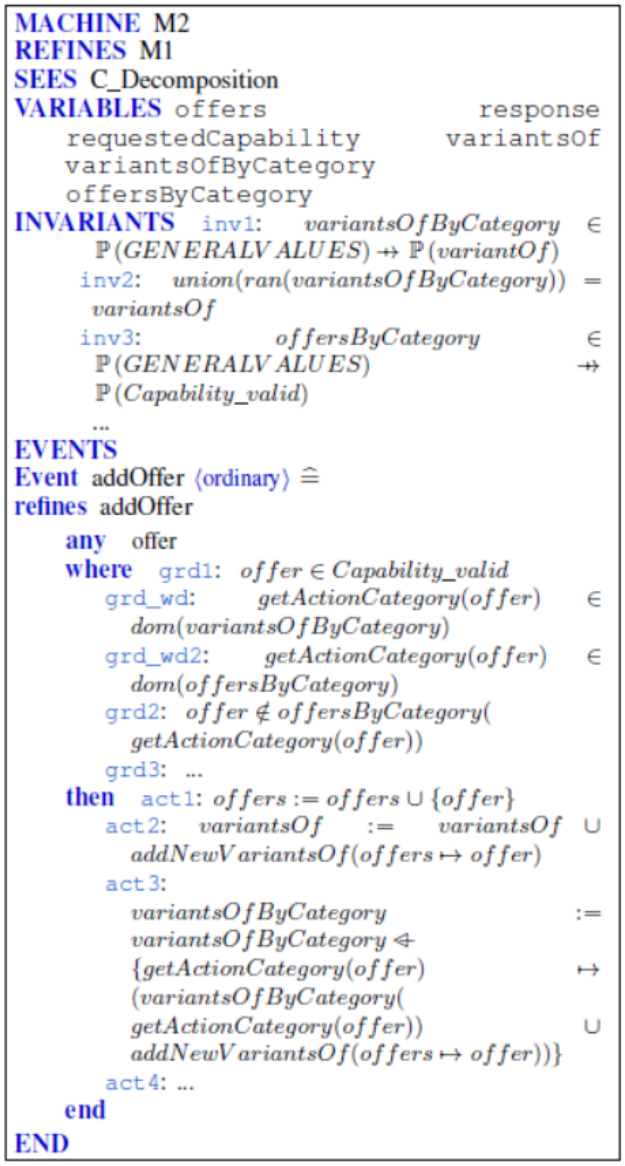


MACHINE M1
REFINES M0
SEES C_Basic_Behavior
VARIABLES offers response requestedCapability variantsOf
INVARIANTS inv1: $offers \subseteq Capability\_valid$
inv2: $requestedCapability = \varnothing \vee requestedCapability \in Capability\_valid$
inv3: $variantsOf = \{c1 \mapsto c2 | c1 \in offers \wedge c2 \in offers \wedge c1 \mapsto c2 \in variantOf\}$
inv4a: $requestedCapability \in offers \Rightarrow response = \{requestedCapability\}$
inv4b: $requestedCapability \notin offers \Rightarrow response = \{offer | offer \in offers \wedge requestedCapability \mapsto offer \in variantOf\}$
EVENTS
Event addOfferWhenRequestIsEqualToOffer ⟨ordinary⟩ $\widehat{=}$
refines addOffer
any offer
where grd1: $offer \in Capability\_valid$
grd2: $offer \notin offers$
grd3: $offer = requestedCapability$
then act1: $offers := offers \cup \{offer\}$
act2: $response := \{offer\}$
act3: $variantsOf := variantsOf \cup addNewVariantsOf(offers \mapsto offer)$
end
...
END

Fig. 11. Snapshot of the Event-B M1 defining the first refinement.

- removeOfferWhenOfferIsInResponse-AndEqualsRequest, removeOfferWhenOfferIs-InResponseAnd-NotEqualToRequest, and removeOffer: These events differentiate between removing an offer that equals the current request, that is part of the response but not equal, or that is unrelated.
- getOffersForNewRequestWhenRequest-IsInOffer and getOffersForNewRequestWhenRequ-estIsNotInOffer: These events handle the case where the request is already in the repository (response is the request itself), or not (response is the set of variantOf offers).

- M2 – Second refinement: (Fig. 12) This refinement introduces the notion of service categorization through subgraphs. Based on the actionCategory property, mandatory in each capability as per the Capability Model, offers are grouped into subgraphs, each corresponding to a distinct category of service. This categorization enhances efficiency by allowing discovery to be restricted to the relevant subgraph. Our primary objective in this refinement is to prove that the global graph, represented by offers and variantsOf, is equivalent to the union of these subgraphs. Additionally, we verify that these subgraphs are disjoint.



MACHINE M2
REFINES M1
SEES C_Decomposition
VARIABLES offers response requestedCapability variantsOf variantsOfByCategory offersByCategory
INVARIANTS inv1: $variantsOfByCategory \in \mathbb{P}(GENERALVALUES) \rightarrow\!\!\!\!\!\shortmid\; \mathbb{P}(variantOf)$
inv2: $union(ran(variantsOfByCategory)) = variantsOf$
inv3: $offersByCategory \in \mathbb{P}(GENERALVALUES) \rightarrowtail\!\!\!\!\shortmid\; \mathbb{P}(Capability\_valid)$
...
EVENTS
Event addOffer ⟨ordinary⟩ $\widehat{=}$
refines addOffer
any offer
where grd1: $offer \in Capability\_valid$
grd_wd: $getActionCategory(offer) \in dom(variantsOfByCategory)$
grd_wd2: $getActionCategory(offer) \in dom(offersByCategory)$
grd2: $offer \notin offersByCategory(getActionCategory(offer))$
grd3: ...
then act1: $offers := offers \cup \{offer\}$
act2: $variantsOf := variantsOf \cup addNewVariantsOf(offers \mapsto offer)$
act3: $variantsOfByCategory := variantsOfByCategory \lhd\!\!\!- \{getActionCategory(offer) \mapsto (variantsOfByCategory(getActionCategory(offer)) \cup addNewVariantsOf(offers \mapsto offer))\}$
act4: ...
end
END

Fig. 12. Snapshot of the Event-B M2 defining the second refinement.

Technically, we define new variables including offersByCategory and variantsOfByCategory. They are linked to the global structure(e.g the graph) through the gluing invariants:

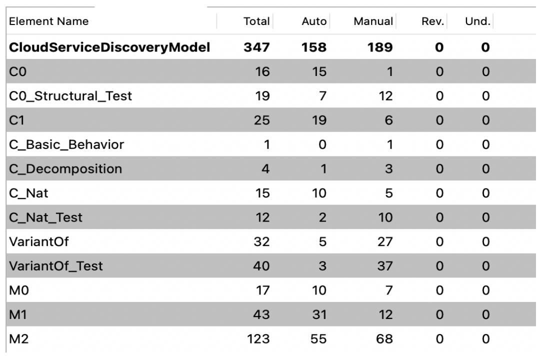
$$offers = \bigcup offersByCategory,$$

$$variantsOf = \bigcup variantsOfByCategory.$$

B. Validation

In this section, we validate our CSD model by leveraging twos kinds of validations described as follows.

1) Proof obligation: As shown in Fig. 13, for the complete model, a total of 347 proof obligations were generated. Among these, 158 were discharged automatically, while 189 required manual intervention. Notably, 123 obligations were associated specifically with the machine `M2`, of which 68 were discharged manually.



Element Name	Total	Auto	Manual	Rev.	Und.
CloudServiceDiscoveryModel	**347**	**158**	**189**	**0**	**0**
C0	16	15	1	0	0
C0_Structural_Test	19	7	12	0	0
C1	25	19	6	0	0
C_Basic_Behavior	1	0	1	0	0
C_Decomposition	4	1	3	0	0
C_Nat	15	10	5	0	0
C_Nat_Test	12	2	10	0	0
VariantOf	32	5	27	0	0
VariantOf_Test	40	3	37	0	0
M0	17	10	7	0	0
M1	43	31	12	0	0
M2	123	55	68	0	0

Fig. 13. Proof statistics view from the Rodin platform showing automatic and manual discharge across refinement levels.

2) Validation through interaction scenarios: Although all proof obligations associated with the formal Event-B model were successfully discharged, it remained necessary to validate the system's behavior under dynamic operations such as service insertion, removal, and user request handling. However, due to the complexity of the model, particularly the expressive axioms involving property value types and specification relations, ProB, which supports model animation, was unable to complete execution in our experiments. To address this limitation, we employed a manual, scenario-based validation strategy. For each interaction, the resulting system state was manually derived by applying the `action` clause of the corresponding Event-B event to the previous state. Because all events were formally verified to preserve the model's invariants, each state in the scenario is guaranteed to be consistent with the formally defined behavior. This process effectively simulates the system's dynamic execution and constitutes a valid behavioral validation trace grounded in the provably correct model.

For this purpose, we reuse the same structure as in the motivating example presented earlier. The capabilities C_1, C_2, C_3, C_4 , described below, follow the same structure and naming convention as in the logistics example, are now adapted to a cloud computing context. Each capability is compliant with the formally defined `Capability_valid` structure, and thus their structural correctness is already formally verified. In this context, we simulate a scenario involving a service provider p, who adds and removes service offers, and two users, u_1 and u_2, who send discovery requests.

- C_1: An infrastructure as a service(IaaS) cloud offering providing a standard virtual machine (VM) in the AWS[3] limited to the Europe region .

$$C_1 = \{\texttt{actionCategory} \mapsto \texttt{compute},\\ \texttt{region} \mapsto \texttt{Europe},\ \texttt{provider} \mapsto \texttt{aws}\}$$

- C_2: A specialized offer using `t3` instance family.

$$C_2 = \{\texttt{actionCategory} \mapsto \texttt{compute},\\ \texttt{region} \mapsto \texttt{Europe},\ \texttt{provider} \mapsto \texttt{aws},\\ \texttt{instanceType} \mapsto \texttt{t3}\}$$

- C_3: A further specialized offer targeting specific zones (e.g., `eu-west-2` for London and `eu-west-3` for Paris).

$$C_3 = \{\texttt{actionCategory} \mapsto \texttt{compute},\\ \texttt{region} \mapsto \{\texttt{eu-west-2}, \texttt{eu-west-3}\},\\ \texttt{provider} \mapsto \texttt{aws}\}$$

- C_4: A variant of C_2 that introduces a Static Solid Storage(SSD)

$$C_4 = \{\texttt{actionCategory} \mapsto \texttt{compute},\\ \texttt{region} \mapsto \texttt{eu-west-2},\ \texttt{provider} \mapsto \texttt{aws},\\ \texttt{instanceType} \mapsto \texttt{t3.micro},\\ \texttt{storageType} \mapsto \texttt{SSD}\}$$

These capabilities are all defined in accordance with the verified Event-B model, where each capability is a set of property entries satisfying domain-specific constraints (e.g., $\texttt{region} \subseteq \texttt{GeographicalLocation}$). As such, no additional assumptions regarding validity are required: the model ensures correctness by construction.

This real-world example illustrates the direct applicability of our approach to cloud computing environments, demonstrating how `variantOf` relations support capability specialization and service discovery over a formally verified and semantically structured service repository.

The scenario proceeds as follows:

1) The provider adds capability C_1.
2) The provider adds capability C_2.
3) User u_1 submits a request for C_2.
4) User u_2 submits a request for C_4.
5) The provider adds capability C_3.
6) The provider removes capability C_2.
7) The provider re-adds capability C_2.

Each step in this sequence modifies the system state by triggering an Event-B event. The updated values of key variables such as `offers`, `variantsOf`, `response`, and `requestedCapability` are depicted in Fig. 14. These transitions are shown to preserve the model invariants, thereby demonstrating the behavioral correctness of the system.

VI. Results and Discussion

Our Event-B model for cloud service discovery (CSD) has been developed incrementally. In Section IV, we proposed the Event-B model for the Capability Model. The proposed model not only covers all types of properties in the Capability Model, but also goes beyond by formalizing the variantOf relation

[3]https://aws.amazon.com

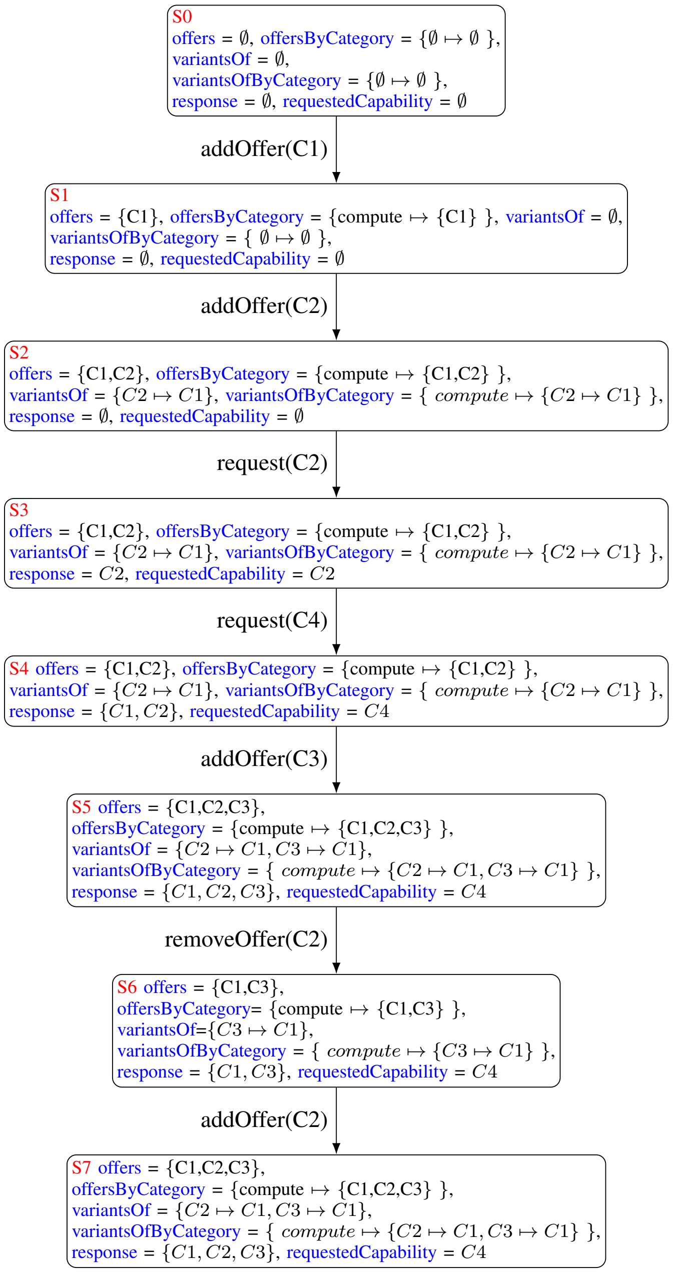

S0
offers = ∅, offersByCategory = {∅ ↦ ∅ },
variantsOf = ∅,
variantsOfByCategory = {∅ ↦ ∅ },
response = ∅, requestedCapability = ∅
addOffer(C1)
S1
offers = {C1}, offersByCategory = {compute ↦ {C1} }, variantsOf = ∅,
variantsOfByCategory = { ∅ ↦ ∅ },
response = ∅, requestedCapability = ∅
addOffer(C2)
S2
offers = {C1,C2}, offersByCategory = {compute ↦ {C1,C2} },
variantsOf = {C2 ↦ C1}, variantsOfByCategory = { compute ↦ {C2 ↦ C1} },
response = ∅, requestedCapability = ∅
request(C2)
S3
offers = {C1,C2}, offersByCategory = {compute ↦ {C1,C2} },
variantsOf = {C2 ↦ C1}, variantsOfByCategory = { compute ↦ {C2 ↦ C1} },
response = C2, requestedCapability = C2
request(C4)
S4 offers = {C1,C2}, offersByCategory = {compute ↦ {C1,C2} },
variantsOf = {C2 ↦ C1}, variantsOfByCategory = { compute ↦ {C2 ↦ C1} },
response = {C1, C2}, requestedCapability = C4
addOffer(C3)
S5 offers = {C1,C2,C3},
offersByCategory = {compute ↦ {C1,C2,C3} },
variantsOf = {C2 ↦ C1, C3 ↦ C1},
variantsOfByCategory = { compute ↦ {C2 ↦ C1, C3 ↦ C1} },
response = {C1, C2, C3}, requestedCapability = C4
removeOffer(C2)
S6 offers = {C1,C3},
offersByCategory= {compute ↦ {C1,C3} },
variantsOf={C3 ↦ C1},
variantsOfByCategory = { compute ↦ {C3 ↦ C1} },
response = {C1, C3}, requestedCapability = C4
addOffer(C2)
S7 offers = {C1,C2,C3},
offersByCategory = {compute ↦ {C1,C2,C3} },
variantsOf = {C2 ↦ C1, C3 ↦ C1},
variantsOfByCategory = { compute ↦ {C2 ↦ C1, C3 ↦ C1} },
response = {C1, C2, C3}, requestedCapability = C4

Fig. 14. State transition diagram of the cloud service discovery system under the validation scenario.

between services. Furthermore, we mathematically proved that variantOf satisfies the properties of a partial order relation, enabling the organization of services into a Directed Acyclic Graph (DAG) structure that facilitates discovery. Additionally, we introduced the concept of a valid capability, which supports the structural validation of service definitions.

The model was validated through two types of tests. First, we conducted unit-level validation by encoding test contexts for each component of the model, allowing us to prove structural consistency and the well-definedness of all model elements. Second, we validated the model using our motivating example. This demonstrated that the formal specification is sufficiently expressive to represent the example, confirming the conformity of our formal model with the original Capability Model.

In Section V, we verified a repository-based CSD system built upon the formal Capability Model, used for describing both services and user requests. In this model, we ensured request consistency through the valid capability concept, verified the correct construction of the service repository as a DAG, and validated the correctness of the matching process. Additionally, we confirmed the structured organization of services based on the actions they perform.

We formally proved the correctness of the complete model through 347 proof obligations generated by the Rodin tool, of which 189 were discharged interactively and the rest were discharged automatically. Furthermore, we performed a second validation using a real-world cloud services scenario. Our validation confirmed that our model is mathematically sound, correct by construction, and applicable to cloud service environments.

However, in this initial stage, our approach only supports exact or generalized matching. This restrictive matching strategy is insufficient for practical cloud systems which typically require support for partial satisfaction and ranking mechanisms. Although our model establishes a strong theoretical foundation, further refinement is needed to verify properties that are specific to real-world cloud environments, such as scalability. This involves analyzing of the computational complexity of the matchmaking process, DAG traversal, and repository operations.

VII. Conclusion

In this work, we proposes a formal architecture for capability-centric cloud service discovery, grounded in the Event-B method. The architecture is based on a capability-centric description model called the Capability Model, which supports rich property representations tailored to cloud service characteristics. Our system is repository-based and structured as a directed acyclic graph (DAG) using a formally verified partial order relation denoted as VariantOf. We verified the soundness of this structure by proving the consistency of the service description model and the partial order relation. The behavior of the system was modeled across multiple refinement levels, including a decomposition of services according to the action they perform. 347 proof obligations generated by the Rodin tool were discharged, and a scenario-based validation was conducted to confirm behavioral correctness. However, the matchmaking process currently supports only exact and generalized matching, which may be restrictive in real-world systems. Future work will address this limitation by supporting partial satisfaction and ranking techniques. Furthermore, we plan to formally optimize the repository structure by eliminating transitive edges in the graph.

Acknowledgment

The authors would like to thank the Department of Cooperation and Cultural Action (SCAC) of the French Embassy in Mauritania for its financial support of the three-year research mobility in France.

References

[1] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014, vol. 545.

[2] N. Antonopoulos and L. Gillam, *Cloud computing*. Springer, vol. 51, no. 7.

[3] H. Nabli, R. Ben Djemaa, and I. Amous Ben Amor, "Description, discovery, and recommendation of cloud services: a survey," *Service Oriented Computing and Applications*, vol. 16, no. 3, pp. 147–166, 2022.

[4] M. M. Al-Sayed, H. A. Hassan, and F. A. Omara, "An intelligent cloud service discovery framework," *Future Generation Computer Systems*, vol. 106, pp. 438–466, 2020.

[5] A. Alfazi, Q. Z. Sheng, A. Babar, W. Ruan, and Y. Qin, "Toward unified cloud service discovery for enhanced service identification," in *Service Research and Innovation: 5th and 6th Australasian Symposium, ASSRI 2015 and ASSRI 2017, Sydney, NSW, Australia, November 2–3, 2015, and October 19–20, 2017, Revised Selected Papers 5*. Springer, 2018, pp. 149–163.

[6] G. Di Modica and O. Tomarchio, "Matching the business perspectives of providers and customers in future cloud markets," *Cluster Computing*, vol. 18, pp. 457–475, 2015.

[7] Y. Jiang, D. Tao, Y. Liu, J. Sun, and H. Ling, "Cloud service recommendation based on unstructured textual information," *Future Generation Computer Systems*, vol. 97, pp. 387–396, 2019.

[8] J. Kang and K. M. Sim, "Cloudle: An agent-based cloud search engine that consults a cloud ontology," in *Cloud Computing and Virtualization Conference*. Citeseer, 2010, pp. 312–318.

[9] J. Ka and K. M. Sim, "Ontology-enhanced agent-based cloud service discovery," *International Journal of Cloud Computing*, vol. 5, no. 1-2, pp. 144–171, 2016.

[10] N. Karthikeyan, R. K. RS *et al.*, "Fuzzy service conceptual ontology system for cloud service recommendation," *Computers & Electrical Engineering*, vol. 69, pp. 435–446, 2018.

[11] H. Ma, Z. Hu, K. Li, and H. Zhu, "Variation-aware cloud service selection via collaborative qos prediction," *IEEE Transactions on Services Computing*, vol. 14, no. 6, pp. 1954–1969, 2019.

[12] H. Nabli, R. B. Djemaa, and I. A. B. Amor, "Efficient cloud service discovery approach based on lda topic modeling," *Journal of Systems and Software*, vol. 146, pp. 233–248, 2018.

[13] L. Sun, J. Ma, Y. Zhang, H. Dong, and F. K. Hussain, "Cloud-fuser: Fuzzy ontology and mcdm based cloud service selection," *Future Generation Computer Systems*, vol. 57, pp. 42–55, 2016.

[14] V. Viji Rajendran and S. Swamynathan, "Sd-csr: semantic-based distributed cloud service registry in unstructured p2p networks for augmenting cloud service discovery," *Journal of Network and Systems Management*, vol. 27, pp. 625–646, 2019.

[15] J. Wheal and Y. Yang, "Csrecommender: a cloud service searching and recommendation system," *Journal of Computer and Communications*, vol. 3, no. 6, pp. 65–73, 2015.

[16] M. Zhang, R. Ranjan, A. Haller, D. Georgakopoulos, M. Menzel, and S. Nepal, "An ontology-based system for cloud infrastructure services' discovery," in *8th international conference on collaborative computing: networking, applications and worksharing (CollaborateCom)*. IEEE, 2012, pp. 524–530.

[17] A. Heidari and N. Jafari Navimipour, "Service discovery mechanisms in cloud computing: a comprehensive and systematic literature review," *Kybernetes*, vol. 51, no. 3, pp. 952–981, 2022.

[18] A. Souri, N. J. Navimipour, and A. M. Rahmani, "Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review," *Computer Standards & Interfaces*, vol. 58, pp. 1–22, 2018.

[19] K. Klai and H. Ochi, "A formal approach for service composition in a cloud resources sharing context," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 458–461.

[20] Y. Li, S. Zhao, H. Diao, and H. Chen, "A formal validation method for trustworthy services composition," in *2016 International Conference on Networking and Network Applications (NaNA)*. IEEE, 2016, pp. 433–437.

[21] P. Wang, L. Yang, and G. W. Li, "Mobile cloud computing system components composition formal verification method based on space-time pi-calculus," in *International Conference on Cloud Computing*. Springer, 2015, pp. 159–167.

[22] A. Lahouij, L. Hamel, M. Graiet, and B. el Ayeb, "An event-b based approach for cloud composite services verification," *Formal Aspects of Computing*, vol. 32, no. 4, pp. 361–393, 2020.

[23] L. Hamel, M. Graiet, M. Kmimech, M. T. Bhiri, and W. Gaaloul, "Verifying composite service transactional behavior with event-b," in *2011 Seventh International Conference on Semantics, Knowledge and Grids*. IEEE, 2011, pp. 99–106.

[24] M. Barati and R. St-Denis, "An architecture for semantic service discovery and realizability in cloud computing," in *2015 6th International conference on the network of the future (NOF)*. IEEE, 2015, pp. 1–6.

[25] M. Barati, "A formal technique for composing cloud services," *Information Technology and Control*, vol. 49, no. 1, pp. 5–27, 2020.

[26] N. J. Navimipour, "A formal approach for the specification and verification of a trustworthy human resource discovery mechanism in the expert cloud," *Expert Systems with Applications*, vol. 42, no. 15-16, pp. 6112–6131, 2015.

[27] A. Souri and N. J. Navimipour, "Behavioral modeling and formal verification of a resource discovery approach in grid computing," *Expert Systems with Applications*, vol. 41, no. 8, pp. 3831–3849, 2014.

[28] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[29] S. Bhiri, W. Derguech, and M. Zaremba, "Modelling capabilities as attribute-featured entities," in *Web Information Systems and Technologies: 8th International Conference, WEBIST 2012, Porto, Portugal, April 18-21, 2012, Revised Selected Papers 8*. Springer, 2013, pp. 70–85.

[30] W. Derguech and S. Bhiri, "Modelling, interlinking and discovering capabilities," in *2013 ACS International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2013, pp. 1–8.

[31] W. Derguech, S. Bhiri, and E. Curry, "Using ontologies for business capability modelling: describing what services and processes achieve," *The Computer Journal*, vol. 61, no. 7, pp. 1075–1098, 2018.

[32] I. Jerbi, N. Assy, M. Sellami, H. Brabra, W. Gaaloul, S. Bhiri, O. Tirat, and D. Zeghlache, "Enabling multi-provider cloud network service bundling," in *2022 IEEE International Conference on Web Services (ICWS)*. IEEE, 2022, pp. 405–414.

[33] G. De Giacomo, F. Patrizi, and S. Sardina, "Automatic behavior composition synthesis," *Artificial Intelligence*, vol. 196, pp. 106–142, 2013.

[34] J. Kang and K. M. Sim, "Cloudle: a multi-criteria cloud service search engine," in *2010 IEEE Asia-Pacific Services Computing Conference*. IEEE, 2010, pp. 339–346.

[35] J. K and K. M. Sim, "Cloudle: an ontology-enhanced cloud service search engine," in *International Conference on Web Information Systems Engineering*. Springer, 2010, pp. 416–427.

[36] J. Kang and K. M. Sim, "Ontology and search engine for cloud computing system," in *Proceedings 2011 International Conference on System Science and Engineering*. IEEE, 2011, pp. 276–281.

[37] J. Ka and K. M. Sim, "Towards agents and ontology for cloud service discovery," in *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. IEEE, 2011, pp. 483–490.

[38] K. M. Sim, "Agent-based cloud computing," *IEEE transactions on services computing*, vol. 5, no. 4, pp. 564–577, 2011.

[39] S. Gong and K. M. Sim, "Cb-cloudle and cloud crawlers," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*. IEEE, 2014, pp. 9–12.

[40] A. Alfazi, T. H. Noor, Q. Z. Sheng, and Y. Xu, "Towards ontology-enhanced cloud services discovery," in *Advanced Data Mining and Applications: 10th International Conference, ADMA 2014, Guilin, China, December 19-21, 2014. Proceedings 10*. Springer, 2014, pp. 616–629.

[41] T. H. Noor, Q. Z. Sheng, A. Alfazi, A. H. Ngu, and J. Law, "Csce: a crawler engine for cloud services discovery on the world wide web," in *2013 IEEE 20th International Conference on Web Services*. IEEE, 2013, pp. 443–450.

[42] M. Á. Rodríguez-García, R. Valencia-García, F. García-Sánchez, and J. J. Samper-Zapater, "Ontology-based annotation and retrieval of services in the cloud," *Knowledge-based systems*, vol. 56, pp. 15–25, 2014.

[43] K. Boukadi, M. Rekik, M. Rekik, and H. Ben-Abdallah, "Fc4cd: a new soa-based focused crawler for cloud service discovery," *Computing*, vol. 100, pp. 1081–1107, 2018.

[44] A. Q. Md, V. Varadarajan, and K. Mandal, "Efficient algorithm for identification and cache based discovery of cloud services," *Mobile Networks and Applications*, vol. 24, no. 4, pp. 1181–1197, 2019.

[45] I. Jerbi and S. Bhiri, "Definition and induction of a specification order relation between capabilities," in *2021 IEEE International Conference on Services Computing (SCC)*. IEEE, 2021, pp. 126–133.

[46] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in event-b," *International journal on software tools for technology transfer*, vol. 12, pp. 447–466, 2010.

[47] P. Oaks, A. H. Ter Hofstede, and D. Edmond, "Capabilities: Describing what services can do," in *Service-Oriented Computing-ICSOC 2003: First International Conference, Trento, Italy, December 15-18, 2003. Proceedings 1*. Springer, 2003, pp. 1–16.

[48] S. Bhiri, W. Derguech, and M. Zaremba, "Web service capability meta model." in *WEBIST*, 2012, pp. 47–57.