Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 96 (2016) 617 – 626

Procedia
Computer Science

The 20th International Conference on Knowledge Based and Intelligent Information and Engineering Systems, York, United Kingdom

Modelling Functional Behavior of Event-Based Systems: A Practical Knowledge-Based Approach

Fahim T. Imam*, Thomas R. Dean

School of Computing, Queen's University, Kingston, Ontario, K7L3N6, Canada

Abstract

Functional behavior is considered to be the most basic, yet a critical notion in order to determine the characteristics of a system. However, how to reason about the functional behavior of a system in a systematic manner, is mostly limited by our cognitive processing abilities. While the UML-based behavior models can support a visual conceptualization of the functional behavior, they lack the rigorous, machine-processable reasoning capabilities. In this paper, we present a practical, knowledge-based approach to model the functional behavior that incorporates the notions of Commonsense Reasoning and Functional Reasoning over its core defining aspects. We demonstrate our approach with a detailed example, along with a set of use case scenarios. The main motivation behind this work was to develop a rigorous, logic-based approach to verify the levels of functional consistencies between cross-platform event-based systems. The focus of this paper, however, is to present the representational facility that can be utilized for the consistency validation system. While we provide a brief overview of the consistency validation system in this paper, a separate article will be dedicated for the comprehensive overview of the validation system itself.

Peer-review under responsibility of KES International

Keywords: Functional Reasoning; Systems Behaviour; Event-Driven Systems; Ontologies;

1. Introduction

Representing a system in terms of its functional behavior is a critical practice for any Engineering discipline. Without an effective formal representation, systematic reasoning about the behavior of a complex system would always be limited by our cognitive processing abilities. While the UML-based behavior modeling can support a visual conceptualization, they lack the automated reasoning capabilities, due to their semi-formal nature of representation. While the significance of formal method based systems, e.g., model-checkers, is indisputable for developing reliable software systems, their actual use within the software engineering community is still quite rare. While the reasoning mechanisms of these formal systems are mostly automated, the process of modeling the practical application domains based on these systems are often counter-intuitive and requires a considerable investment of time and expertise. Therefore, in practice, the powerful notion of formal methods are barely utilized in software engineering projects.

[*] Corresponding author. Tel.: +1-613-331-4657.
E-mail address: fahim.imam@queensu.ca

Peer-review under responsibility of KES International
doi:10.1016/j.procs.2016.08.243

In this paper, we present a practical, knowledge-based approach of functional behavior modeling that utilizes a rigorous, ontological representation of event-based systems. The representational facility is mostly based on the theory of *action*, *events*, and *change*, as understood in the studies of the AI-based Commonsense Reasoning[10 2], Functional Reasoning[3], and the Event Calculus[11]. Our approach promotes an enhanced comprehension and control over the following defining aspects of event-based systems: (a) the allowable set of events; (b) the allowable flow of events; (c) event locations and interfaces; and, (d) the agents interacting with the interfaces. We have carefully analyzed these latter aspects and handcrafted an effective ontological representation that can be utilized within the context of functional behavior modelling. We have engineered the representation in such a way so that the ontology can adequately utilize the existing inference engines for its required reasoning services.

The ontology that we developed facilitates us to model a system's behavior using a set of intuitive natural language expressions for human comprehension as supported by the ontology's conceptual semantics. The systems modelled based on the ontology can also promote a certain level of intelligence within the modeled entities themselves. When invoked by an automated reasoner on its instances, the ontology can infer the implicit, logical consequences of the explicitly specified functional aspects in a number different ways. These include, but are not limited to, automated functional reasoning on the combination and permutation of the relevant functional aspects, discovering new interrelations between different events, agents, and interfaces, checking the functional consistencies between evolving systems, and so forth. Our approach can be useful for those systems that require automated reasoning in order to characterize, compare, and ultimately, comprehend the functional behavior of the systems at various levels of useful abstractions. The rest of the paper is organized as follows. In Section 2, we provide the background, motivation, and discuss the key notions relevant to the remaining sections of this paper. Section 3 provides an overview of our representational facility. We will go through a demonstration of our functional modeling approach in Section 4. In Section 5, we discuss the use case scenarios of our approach. Finally, we conclude this paper in Section 6.

2. Background and Related Work

2.1. The Key Motivation

The key motivation behind this work is to develop a rigourous, formal logic-based approach to verify the levels of functional consistencies between cross-platform event-based systems. Supporting multiple, heterogeneous platforms for the modern systems, e.g., the smart phones and tablet computing devices, is a common requirement for any large-scale software engineering projects, e.g., airlines reservation systems, banking systems, social-networking systems, and so forth. When dealing with the functional behavior of these systems, an effective software engineering approach must tackle the following key challenges: (a) validating the level of functional consistencies between the cross-platform application systems; (b) maintaining a consistent co-evolution of the cross-platform functionalities based on the evolving requirements of the system's domain; and, (c) comprehending the functional changes that can be both human comprehensible and machine processable. The main focus of this paper is to provide an adequate representational facility that can promote a machine-processable, intelligent decision-support mechanism for the kinds of functional reasoning that is required in order to mitigate or potentially overcome these latter challenges.

Despite the unavoidable heterogeneity issues with multiple platforms, the systems developed for those platforms must confirm that their intended set of functionalities, are in fact, consistently implemented across the platforms. The tasks of maintenance in these systems can be quite expensive and challenging. An effective maintenance for the large-scale evolving systems must require a rigorous strategy for consistency management and change propagation across the overall system components. While the importance of software testing cannot be denied as part of the maintenance process, generating all possible test cases, and tracing the solution fragment for negative results, can be quite overwhelming. Contrary to software testing, we considered a more systematic, formal approach for the consistency management. However, our approach can help categorizing the set of test cases that must be executed in order to validate the functional consistencies between the cross-platform and evolving systems.

The core idea of our approach is to utilize the powerful notion of ontologies in order to mitigate the heterogeneity issues among the systems that are intended to have a set of identical functional goals. By characterizing the functional behavior into an ontology, we can have an effective mechanism to validate the levels of functional consistency among the systems. Since we are dealing with the event-based systems like mobile apps, the focus of the representation should be the functionalities that can be observed from its User Interface (UI) elements. Since the events associated with the UI elements are indicative of the system's behavior[7], these elements should provide the key source of knowledge regarding the functional models of the comparing systems. A set of such functional models should be captured at a level of abstraction that is suitable for our purpose of validating cross-platform, event-based functional consistencies between the comparing systems. This paper is about of the representational facility that we developed for our consistency validation system. While we provide a brief, high-level overview of the validation system in Section 5, a separate article will be dedicated in the future where we will present the actual validation system in a detailed, systematic manner.

2.2. The Notion of Ontologies

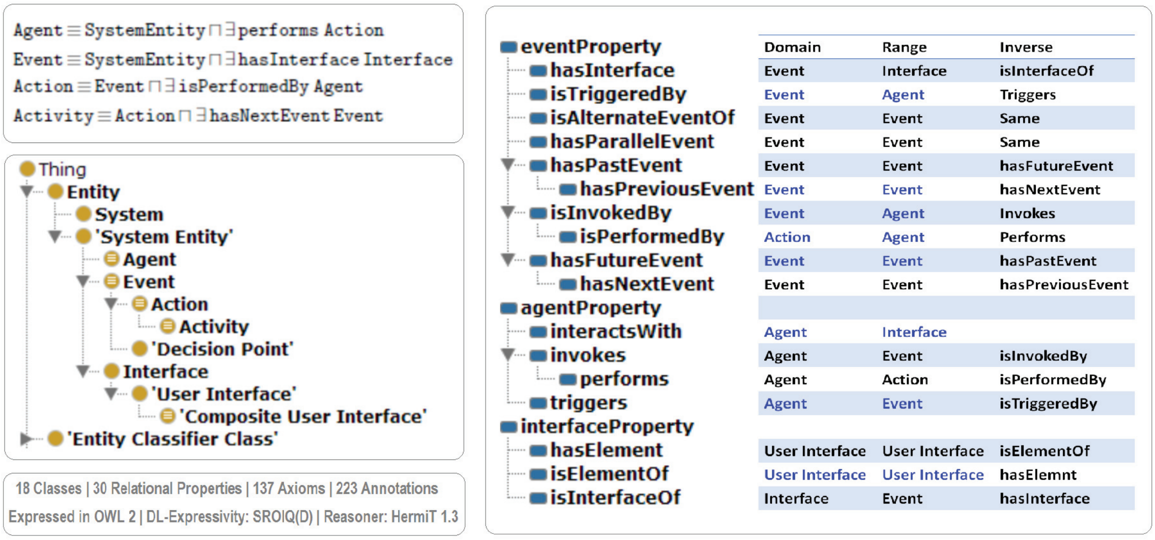
The philosophical term *ontology* was first adapted to Computer science by Gruber[5] as "a formal explicit specification of a shared conceptualization" for the AI community. An ontology represents the concepts within a domain and specifies how the concepts are related to each other through a set of logical axioms. An ontological knowledgebase, K can be defined as the 4-tuple, $K{=}(C, R, A, I)$, where, C represents the set of Concepts or Classes within a domain of interest, R represents the set of binary relations between two classes, A represents the set of logical Axioms to further specialize the classes, and, I represents the set instances of the Classes, i.e., the ground-level individual objects. The logical axioms in an ontology are typically expressed in a formal language such as in Web Ontology Language (OWL)[8]. Developed based on the fragments of First Order Logic (FOL), OWL Description Logic (OWL-DL) has become the de-facto language for developing modern ontologies. An ontologically represented system can provide intelligent decision support mechanisms for its formally expressed body of knowledge. One of the most powerful features of an ontology is that it provides the logical means to express the explicit knowledge of a conceptual domain, from which, the implicit, new knowledge can be inferred through the logical reasoners or inference engines[12 1].

2.3. The Notion of Functional Reasoning

Functional reasoning refers to the theories and techniques to formalize the functional behavior of a system in terms of the following key questions: (a) What are the set of functions that should exist on the desired system? (b) What are the relations that exist between the functions? (c) How to explain the relationships between the functions and the system's artifacts? (d) How to derive the purpose of the artifacts in terms of their functionalities? According to Far et al.[3], the notion of functional reasoning has the following constituent parts: (a) An ontology to describes the functional entities of a system; (b) A representation formalism to model the interactions of the entities, and (c) A reasoning mechanism to infer the implicit functions of the entities. As suggested by Far et al.[3] the ultimate goal of functional reasoning is to achieve the commonsense reasoning about the functionality of the desired system. Commonsense reasoning is one of the prominent goals of AI research which deals with the challenges of representing computing systems in terms of our common, everyday knowledge[10]. We consider the notion of commonsense reasoning to be the basis for our functional reasoning mechanism. Refer to Far et al.[3] for a detailed survey of different functional reasoning theories, perspectives, and their targeted problem domains.

2.4. The Notion of Action, Events, and Change

The theory of action, events, and change is one of the critical areas of commonsense reasoning[10]. Based on the understanding by Davis et al.[2], we consider the following constraints about the systems that involve the notions of events and change: (1) Events are atomic, i.e., they are occurrents and do not travel through time; (2) Every change in the world is caused by an event; (3) Events are deterministic; i.e., the state of the world at the end of an event can be fully determined by the state of the world at the beginning; (4) In order to reason about events, a reasoner simply needs to consider the state of the world at the beginning and the end of the event; the intermediate states should be ignored; and, (5) The entire relevant state of the world at the beginning and all the exogenous events are knowable. The formalisms such as the *event calculus*[11 9] and *situation calculus*[6] are solely dedicated into the concern of representing actions and their effects on an environment using formal logic. The contributions mentioned in this section formed the key logical basis of our proposed representational facility as presented in the next section.



Agent ≡ SystemEntity ⊓ ∃performs Action

Event ≡ SystemEntity ⊓ ∃hasInterface Interface

Action ≡ Event ⊓ ∃isPerformedBy Agent

Activity ≡ Action ⊓ ∃hasNextEvent Event

- Thing
  - Entity
    - System
    - 'System Entity'
      - Agent
      - Event
        - Action
          - Activity
        - 'Decision Point'
      - Interface
        - 'User Interface'
          - 'Composite User Interface'
  - 'Entity Classifier Class'

18 Classes | 30 Relational Properties | 137 Axioms | 223 Annotations

Expressed in OWL 2 | DL-Expressivity: SROIQ(D) | Reasoner: HermiT 1.3

	Domain	Range	Inverse
eventProperty			
hasInterface	Event	Interface	isInterfaceOf
isTriggeredBy	Event	Agent	Triggers
isAlternateEventOf	Event	Event	Same
hasParallelEvent	Event	Event	Same
hasPastEvent	Event	Event	hasFutureEvent
hasPreviousEvent	Event	Event	hasNextEvent
isInvokedBy	Event	Agent	Invokes
isPerformedBy	Action	Agent	Performs
hasFutureEvent	Event	Event	hasPastEvent
hasNextEvent	Event	Event	hasPreviousEvent
agentProperty			
interactsWith	Agent	Interface	
invokes	Agent	Event	isInvokedBy
performs	Agent	Action	isPerformedBy
triggers	Agent	Event	isTriggeredBy
interfaceProperty			
hasElement	User Interface	User Interface	isElementOf
isElementOf	User Interface	User Interface	hasElemnt
isInterfaceOf	Interface	Event	hasInterface

Fig. 1. The EFBO Ontology: Key Classes and Properties.

3. The Proposed Approach: An Overview

In this section, we discuss the concepts relevant to the notion of *events*, along with their logical relationships within the context of functional reasoning on event-based systems. By the end of next section, we should have a solid understanding of the ways we can model a series of interactions among a set of functional entities using our proposed representational facility.

3.1. The Ontological Representation

In order to represent the functional behavior of event-based systems, we have developed the **E**vent-Based **F**unctional **B**ehavior **O**ntology (EFBO, `http://cs.queensu.ca/~imam/efbo.html`). The ontology is designed based on the key notions of events in commonsense reasoning, event calculus, and functional reasoning, along with a series of experiments. Figure 1 presents the key classes and properties (relations) of the EFBO ontology. On the left, we have the logical *is-a* hierarchy of the key classes. On the right, we have a set of asserted relational properties, along with their domain and range restrictions. The domain and range in blue fonts are inferred automatically by invoking a reasoner. Most of the EFBO properties have their corresponding inverse properties which serve a critical role while describing the property chains, as well as the instances of the EFBO ontology. The EFBO ontology can be represented in terms of the following sets of classes and properties.

- A set of classes, $C = \{E, G, I\}$, where, E represents the set of *Event* instances, E=$\{e_1, e_2, .., e_n\}$; G is the set of *Agent* instances, G=$\{g_1, g_2, .., g_n\}$; and, I represents the set of $Interface$ instances, I=$\{i_1, i_2, .., i_n\}$.
- A set of relations, R=$\{R(e, g), R(e, i), R(e_1, e_2)\}$, where, $R(e, g)$ represents the set of relations between an instance of an event, $Event(e)$ and an instance of an agent, $Agent(g)$, e.g., $isPerformedBy$; $R(e, i)$ represents the set of relations between an $Event(e)$ and an $Interface(i)$, e.g., $hasInterface$; and, $R(e_1, e_2)$ represents the set of relations between two *Event* instances, e_1 and e_2; e.g., $hasNextEvent$.

The EFBO is engineered in a way so that we can model the instances of the classes in C in terms of their associated relations in R. The core classes in the EFBO are the *Event*, *Agent*, and the $Interface$ classes. All the other entities within the EFBO are logically based on these three classes, either through direct assertions or through inference.

- ***Event.*** An event within a system is an occurrent - i.e., each of the events in a system must have a single time-point of occurrence. We use the relational properties $hasNextEvent$ and its inverse $hasPreviousEvent$ between two events in order to mark the occurrence of an event. These relations are defined as the non-transitive relations between two events; i.e., they refer to the immediate next and the immediate previous event of an event's occurrence. The properties $hasFutureEvent$ and its inverse $hasPastEvent$ are specified as transitive. These transitive properties are defined as the super properties of the chain of *hasNextEvent* and the chain of *hasPreviousEvent* properties, respectively. If an event x has a next event y, and y has a next event z, then we can infer the fact that z has a past event x. These properties allow an event to be reasoned based on any set of past or future events including the actions and activities associated with those events. An event within the EFBO must have an associated system or environment with a set of interfaces. The property $isAlternateEventOf$ can be used to specify the relation between two event instances that are mutually exclusive i.e., the instances that hold this property must not occur in parallel. The symmetric properties $isAlternateEventOf$ and *hasParallalEvent* are therefore specified as disjoint with each other.

- ***Action.*** The concept of *Action* within the EFBO is specified as an event that is performed by an agent. However, the action cannot be performed directly without invoking the interface of an event. Using the powerful notion of OWL property chain, we have defined the relational property $isPerformedBy$ as the super property of the following chain of properties: $hasInterface\ o\ interactsWith$; i.e., if an $Event(e)$ has an $Interface(i)$ and the interface i interacts with an $Agent(g)$, then the event e becomes an action which is performed by the agent g. The relation *isInvokedBy* refers to the super property of the relation $isPerformedBy$. The former is a relation between an event and and agent, and the latter is between an action and an agent. The set of instances that holds these two properties can only be inferred and must not be asserted directly.
- ***Activity.*** An *Activity* is understood to be an *Action* (i.e., an event performed by an agent) that is followed by a next, successive event. In order to activate an event, the event must be triggered by an agent through the system's interface. We define the $isTriggeredBy$ relation between an event and an agent as the super property of the following chain of properties: $hasPreviousEvent\ o\ isPerformedBy$; i.e., if $Event(e_2)$ has a previous event $Event(e_1)$, and the event e_1 is performed by $Agent(g)$, then the event e_2 is triggered by the agent g.

It should be noted that an event within the EFBO represents an activity that occurs at a particular time. Unlike activities, an event does not travel over time; instead, an event just occurs. Within the context of the EFBO, a useful analogy to distinguish the differences between the concepts of activity and event is the following: we can think of the concept of activity as an ongoing movie which has a beginning, followed by a series of scenes, and an ending. An event, on the other hand, can be thought as a snapshot or a picture of a particular moment in that movie. Analogous to the notion of a movie being nothing more than a series of pictures, the notion of activity can be thought as a series of events. In this latter sense, the beginning of an activity can be an event, a completed sub-activity within an activity can also be considered as an event. Essentially, this latter analogy makes the EFBO concepts of *Event* and *Activity* to correspond to the very notions of SNAP and SPAN as endorsed by the Basic Formal Ontology (BFO)[4].

3.2. The EFBO-Based Functional Modelling

The EFBO ontology can represent a system's change of events along with their agents and interfaces in a rigorous, logical manner. By definition, each event must have an associated point in time. The concept of time is understood as the continued progression of events that occur in an irreversible succession from the past through the present to the future[1]. This latter notion of time and event together forms the Event-Time continuum as depicted in Figure 2 (right). The figure indicates that the event e_n at time t_n is the future event of the event e_{n-1} at time t_{n-1}, and the past event of the next event e_{n+1} at time t_{n+1}. The green and the blue arrows represent an event's temporal connection with its previous and the next events, respectively. While we do not explicitly specify the temporal dimensions of the events, the EFBO has the mechanism to reason about the occurrence of an event through its extensive sup-properties of the *eventProperty* in Figure 1. Using the EFBO, a series of event instances of a system can be modelled and reasoned in terms of different event properties such as the activities of the triggering agents, the interfaces involved, change of interfaces, the causalities, and so forth.

An interaction model among three instances of events is depicted in Figure 2 (left). The event instances are specified as having *hasNextEvent* relations between them. Also, the event instances all have their corresponding interfaces as specified by the *hasInterface* property. The agent instances are specified as having interaction with their corresponding interfaces through the $interactsWith$ relation. Note that, the $interactsWith$ property is defined as a symmetric property between an agent and an interface; i.,g., if $Agent(g1)$ interacts with $Interface(i1)$ that would also mean that the $Interface(i1)$ interacts with the $Agent(g1)$. The relational properties with the dotted arrows in the figure indicate the inferred relationships between two instances. As discussed already, these relations are not meant

[1] We adapted the notion of time from the Oxford Dictionaries which is quite suitable for our purpose of using the existing OWL reasoners without the need of an extensive temporal logic extension.

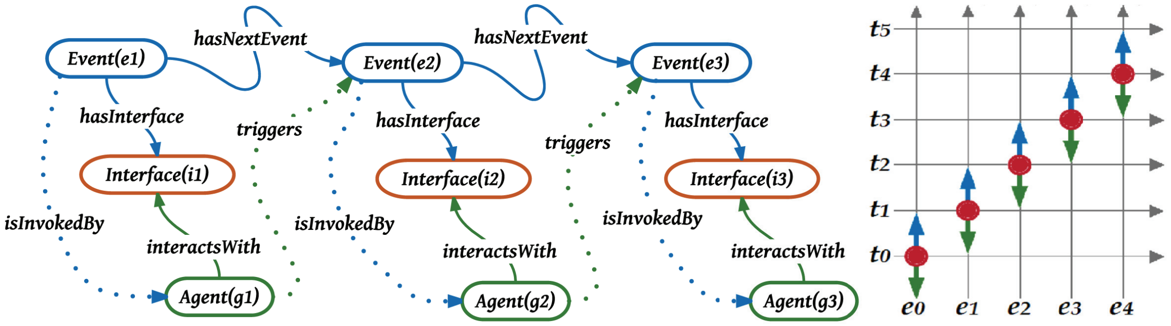


Fig. 2. The *Event(e)-Interface(i)-Agent(g)* Interaction Model (left). The *Event-Time* Continuum (right).

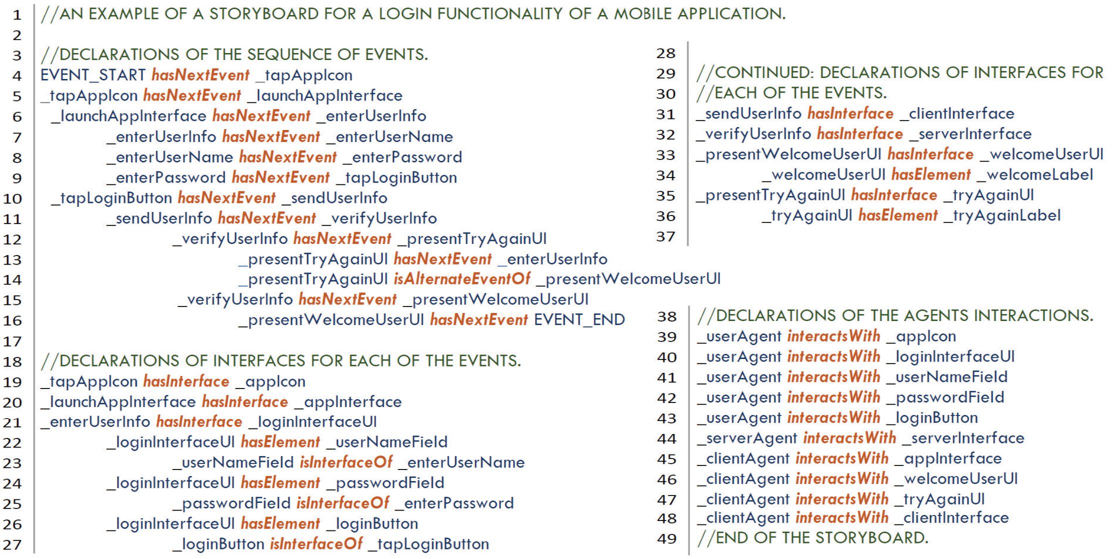
to be asserted directly and must be inferred through automated reasoning. This latter strategy is incorporated in order to model a system's event-based functional behavior in a practical sense; i.e., the agents of such system must not be able to invoke the system's functionality without interacting with the system's designed interfaces. This design choice also promotes more automation for the logical deduction process, which can ultimately reduce the amount of human-induced errors during the modelling process.

Within the perspective of the EFBO, the functionality of a system is understood to be the manifestation of *Event* (i.e., a set of actions, events, and activities). When it comes to deriving the functional behavior of a system, we are simply interested in knowing the following: (1) what are the kind of 'things' (ontologically) that we know are related to the functionality, and (2) how do we formulate the functionality as a set of connections between the 'things' within the ontology model. For example, the functionality of a login button may have three things to observe: (1) login functionality involves some UI component like *login button*; (2) a *login button* is a clickable object for a user agent; and, (3) by clicking the button the user agent triggers some new event to be activated by the client agent. We discuss an example of the actual modelling process using the EFBO's representation facility in the next section.

4. The EFBO-Based Functional Modelling: An Example Demonstration

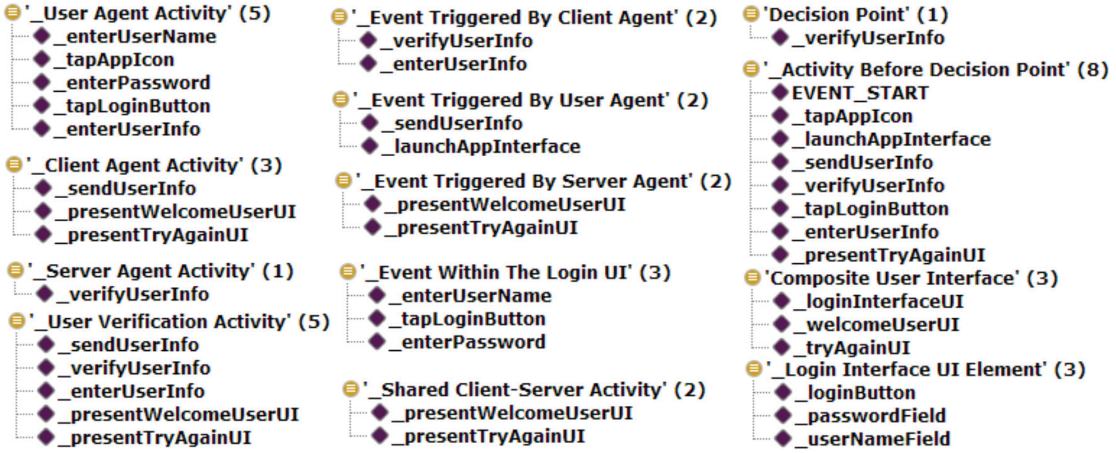
As a demonstration, we observe the typical login functionality of a mobile application in terms of its EFBO representation. While a login system sounds simple enough to follow, it has all the aspects that we need in order to demonstrate the potentials of our representational facilities. Let us first observe the instantiation statements in Figure 3. We can think of the instantiation process as describing the behavior of a system on a storyboard. Based on the EFBO interaction model illustrated in Figure 2, we first declare the sequence of events associated with our login system using the *hasNextEvent* property between each of the events (Lines 5-15). After that, we declare the set of interfaces for each of the declared events using $hasInterface$ relation (Lines 18-37). We also declare the composite UI interfaces using the *hasElement* relation. And finally, we declare the allowable interfaces for each of the agents that need to interact with the system using the *interactsWith* property (Lines 40-48). While the storyboard seems quite simple to follow with only three kinds of properties, all other complex relational properties among the functional entities can be perfectly inferred through automated reasoning. The ideal goal of the storyboard is to state whatever is *explicitly known* about each of the event entities that are involved within a desired system.

One of the powerful features of the EFBO is that it allows the functional entities to be described in a flexible, intuitive manner. For example, we do not have to have direct instantiations for any of the EFBO classes as they can all be perfectly inferred from the specified properties between the instances. The detailed functional activities of an application can be expressed as a series of RDF-like statements of the form $subject - predicate - object$, without any restrictive order. As we can observe from the Figure 3, the EFBO allows us to encode the storyboard in the most natural way possible. The terms prefixed with the underscores are meant to be instances; only exceptions are the predefined EVENT_START and EVENT_END. It should be noted that the sequence of statements within the storyboard can exist in any arbitrary order. No matter in what order they exist, as long as the domain and range constraints are satisfied for each of the relational properties, the EFBO can automatically recognize the actual chain of entities after reasoning.



```
//AN EXAMPLE OF A STORYBOARD FOR A LOGIN FUNCTIONALITY OF A MOBILE APPLICATION.

//DECLARATIONS OF THE SEQUENCE OF EVENTS.
EVENT_START hasNextEvent _tapAppIcon
_tapAppIcon hasNextEvent _launchAppInterface
  _launchAppInterface hasNextEvent _enterUserInfo
        _enterUserInfo hasNextEvent _enterUserName
        _enterUserName hasNextEvent _enterPassword
        _enterPassword hasNextEvent _tapLoginButton
  _tapLoginButton hasNextEvent _sendUserInfo
        _sendUserInfo hasNextEvent _verifyUserInfo
              _verifyUserInfo hasNextEvent _presentTryAgainUI
                    _presentTryAgainUI hasNextEvent _enterUserInfo
                    _presentTryAgainUI isAlternateEventOf _presentWelcomeUserUI
              _verifyUserInfo hasNextEvent _presentWelcomeUserUI
                    _presentWelcomeUserUI hasNextEvent EVENT_END

//DECLARATIONS OF INTERFACES FOR EACH OF THE EVENTS.
_tapAppIcon hasInterface _appIcon
_launchAppInterface hasInterface _appInterface
_enterUserInfo hasInterface _loginInterfaceUI
        _loginInterfaceUI hasElement _userNameField
              _userNameField isInterfaceOf _enterUserName
        _loginInterfaceUI hasElement _passwordField
              _passwordField isInterfaceOf _enterPassword
        _loginInterfaceUI hasElement _loginButton
              _loginButton isInterfaceOf _tapLoginButton

//CONTINUED: DECLARATIONS OF INTERFACES FOR
//EACH OF THE EVENTS.
_sendUserInfo hasInterface _clientInterface
_verifyUserInfo hasInterface _serverInterface
_presentWelcomeUserUI hasInterface _welcomeUserUI
        _welcomeUserUI hasElement _welcomeLabel
_presentTryAgainUI hasInterface _tryAgainUI
        _tryAgainUI hasElement _tryAgainLabel

//DECLARATIONS OF THE AGENTS INTERACTIONS.
_userAgent interactsWith _appIcon
_userAgent interactsWith _loginInterfaceUI
_userAgent interactsWith _userNameField
_userAgent interactsWith _passwordField
_userAgent interactsWith _loginButton
_serverAgent interactsWith _serverInterface
_clientAgent interactsWith _appInterface
_clientAgent interactsWith _welcomeUserUI
_clientAgent interactsWith _tryAgainUI
_clientAgent interactsWith _clientInterface
//END OF THE STORYBOARD.
```

Fig. 3. An example of instantiation statements for the EFBO ontology.



'_User Agent Activity' (5)
- _enterUserName
- _tapAppIcon
- _enterPassword
- _tapLoginButton
- _enterUserInfo

'_Client Agent Activity' (3)
- _sendUserInfo
- _presentWelcomeUserUI
- _presentTryAgainUI

'_Server Agent Activity' (1)
- _verifyUserInfo

'_User Verification Activity' (5)
- _sendUserInfo
- _verifyUserInfo
- _enterUserInfo
- _presentWelcomeUserUI
- _presentTryAgainUI

'_Event Triggered By Client Agent' (2)
- _verifyUserInfo
- _enterUserInfo

'_Event Triggered By User Agent' (2)
- _sendUserInfo
- _launchAppInterface

'_Event Triggered By Server Agent' (2)
- _presentWelcomeUserUI
- _presentTryAgainUI

'_Event Within The Login UI' (3)
- _enterUserName
- _tapLoginButton
- _enterPassword

'_Shared Client-Server Activity' (2)
- _presentWelcomeUserUI
- _presentTryAgainUI

'Decision Point' (1)
- _verifyUserInfo

'_Activity Before Decision Point' (8)
- EVENT_START
- _tapAppIcon
- _launchAppInterface
- _sendUserInfo
- _verifyUserInfo
- _tapLoginButton
- _enterUserInfo
- _presentTryAgainUI

'Composite User Interface' (3)
- _loginInterfaceUI
- _welcomeUserUI
- _tryAgainUI

'_Login Interface UI Element' (3)
- _loginButton
- _passwordField
- _userNameField

Fig. 4. A set of inferred instances for the example system.

Another powerful feature of the EFBO representation is that the entities with the partial or incomplete set of facts can also lead to a certain inferred conclusions by the reasoner. For example, the instance *_tapLoginButton* which has an interface instance *_loginButton* would be simply inferred as an instance of an *Event*. Once we add a new statement that the *_userAgent* interacts with *_loginButton*, the *_tapLoginButton* would be inferred as an instance of an *Action* at that point. An additional statement such as *_tapLoginButton hasNextEvent _sendUserInfo*, would lead the reasoner to infer that the *_tapLoginButton* is an instance of the class *Activity*.

If we have an interface instance $Interface(i)$ that interacts with an agent instance $Agent(g)$ within the EFBO model, the reasoner would infer the fact that $Agent(g)$ has performed an action of an event associated with $Interface(i)$. It should be noted that when stating the facts about *_verifyUserInfo* we don't capture how the exact verification process is going to be implemented. Instead, we only capture the involved entities within the functional flow. However, the EFBO can detect the fact that the *_verifyUserInfo* is a decision point event i.e., an event with multiple alternative next events that are mutually exclusive and must not occur in parallel.

$$\mathrm{\_EventTriggeredByClientAgent} \equiv \mathrm{Event} \sqcap \exists\ \mathrm{isTriggeredBy}\ \{\mathrm{\_clientAgent}\}$$
$$\mathrm{\_EventTriggeredByClientAgent} \sqsubseteq \mathrm{EntityClassifier}$$

$$\mathrm{\_LoginInterfaceElement} \equiv \exists\ \mathrm{isElementOf}\ \{\mathrm{\_loginInterfaceUI}\}$$
$$\mathrm{\_LoginInterfaceElement} \sqsubseteq \mathrm{EntityClassifier}$$

$$\mathrm{\_InterfacePresentedByClientAgent} \equiv \exists\ \mathrm{interactsWith}\ \{\mathrm{\_clientAgent}\}$$
$$\mathrm{\_InterfacePresentedByClientAgent} \sqsubseteq \mathrm{EntityClassifier}$$

$$\mathrm{\_UserAgentActivity} \equiv \exists\ \mathrm{isPerformedBy}\ \{\mathrm{\_userAgent}\}$$
$$\mathrm{\_UserAgentActivity} \sqsubseteq \mathrm{EntityClassifier}$$

$$\mathrm{\_UserVerificationActivity} \equiv \exists\ \mathrm{hasPastEvent}\ \{\mathrm{\_tapLoginButton}\}$$
$$\mathrm{\_UserVerificationActivity} \sqsubseteq \mathrm{EntityClassifier}$$

$$\mathrm{\_InterfaceUsedByUserAgent} \equiv \exists\ \mathrm{interactsWith}\ \{\mathrm{\_userAgent}\}$$
$$\mathrm{\_InterfaceUsedByUserAgent} \sqsubseteq \mathrm{EntityClassifier}$$

Fig. 5. A set of instance-based defined classes.

In order to classify different entities such as the entities specified in Figure 3 statements, we have asserted a set of defined classes into the EFBO ontology. Currently, there are two types of defined classes that we have in the ontology. The first type involves the core EFBO classes such as the *Action*, *Activity*, *Event*, *Agent*, *Interface*, etc. The other types of defined classes, which are defined under the *Entity Classifier Class*, are the application-specific defined classes that involve the specific value of the instances. We distinguish the latter types of classes through their labels prefixed with underscores. The class *Entity Classifier Class* serves as an interface class for the EFBO ontology so that users can define their application-specific classes based on their own classification schema. We provide a subset of our existing defined classes along with their logical axioms in Figure 5 as templates. Additional defined classes can be asserted in order to further classify the instances in other desired ways. Once we pass the asserted statements in Figure 3 along with the EFBO into a reasoner, the reasoner would automatically infer the specific types of instances for each of the entities. Figure 4 presents a set of key classification of instances as obtained by the automated reasoning on the EFBO knowledge models.

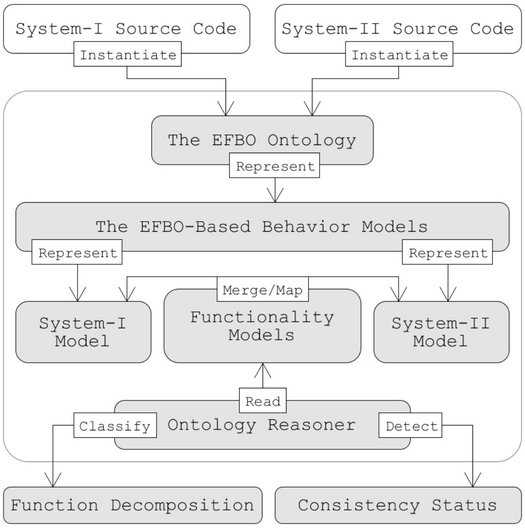
5. Use Case Scenarios

The EFBO-based representation can significantly enhance the requirements specifications for the event-based systems. Compared to the traditional UML-based behavior modelling, the EFBO-based approach would provide a detailed, and rigorous semantics. In addition to representing the generic behavioral model of a system, the EFBO approach can also include the instances of specific source code artifacts such as the UI elements, as part of the behavioral model. Using the EFBO-based approach, the functional behavior models can be viewed, represented, and reasoned over multiple different levels of abstractions. For example, we can ask the reasoner for a set of activities that are associated to achieve a certain functional goal. We can also ask the reasoner about the UI elements involved in those latter activities. Additionally, we can ask the reasoner about the interactions between the client and the server agents within those activities. While the UML-based behavior modeling can provide visual supports for the developer, their role as a machine processable artefact is quite limited. The UML-based approach is only limited to expressing the models into XML-like shareable models, or generating source code structures. Using the UML models, we simply can not ask any of the functional reasoning questions that we have observed in the previous section. Conversely, the EFBO-based models can easily be used to generate the XML-based representation of the UML diagrams, if required.

5.1. The EFBO-Based Functional Reasoning Categories

Extending on the general reasoning categories in event calculus[11] such as the *deductive*, *abductive*, and the *inductive* reasoning, we list the following functional reasoning categories that can be supported by the EFBO representation.

- **Flow of Activities.** This kind of functional reasoning can be used to precisely determine the flow of activities between two functional states. This is a 'What Follows What' (WFW) kind of reasoning question that can be reasoned through the inferred range types of the *hasNextActivity* and *hasPreviousActivity* properties.
- **Action by Action Agents.** This kind of reasoning can be used to verify the exact distribution of activities among different agents. This is a 'Who does What' (WDW) kind of reasoning question that can be asked against the EFBO models.
- **Action by Action Interface.** This kind of reasoning can be used to verify the exact distribution of actions within different UI Interfaces. This is a 'What Happens Where' (WHW) kind of reasoning question that can be asked against the EFBO knowledge models.
- **Shared Activities between Agents**. This kind reasoning can be used to determine the set of shared activities among multiple different agents with a common set of goals. This is a 'Who Shares What' (WSW) kind of reasoning that can be useful to detect any extra, unintended activities between the client and the server agents.



1. Annotate the source code elements of the comparing systems for the core entities of the EFBO.
2. Instantiate the core EFBO entities based on the annotated source code elements. The EFBO will then represent a set of language-independent, comparable functionality models for each of the source systems.
3. Assert the mapping statements between the instantiated entities of the functionality models. This step should represent the merged, overlapping set of functionalities of the comparing systems.
4. Pass the merged functionality models into the automated reasoner in order to derive the implicit classification of the functionalities i.e., derive the functional decompositions.
5. Let the ontology reasoner detect the inconsistent set of functional features, along with the logical reasoning behind the inconsistent cases.
6. Accumulate all the detected inconsistencies into a consistency status report between the systems.

Fig. 6. The EFBO-Base Functional Consistency Verification System.

- **Decision Point Event**. This kind of reasoning can be used to identify the set of events with multiple, alternative next events. This is a 'What Leads to a Decision' (WLD) kind of reasoning question that can be asked against the EFBO knowledge models. In our example, *_verifyUserInfo* is inferred as a decision point since the activity is responded with two alternative events *_presentTryAgainUI* and *_presentWelcomeUserUI* by the server agent. Determining the exact set of activities before and after a decision point can be a critical functional requirement for any event-driven system.

For a system that is modelled based on the EFBO-representation, additional set of reasoning tasks can be composed of the core functional reasoning categories listed above.

5.2. *Validating Functional Consistencies between Systems*

For the systems that are already developed, the EFBO-based representation can be used to annotate and instantiate the program elements that can correspond to the core concepts of event, interface, and agent. For the Java-based applications, we have developed an API called the EFBO-Java Annotator. The API can be used to add EFBO-specific extra annotations within the Java source codes. Using these extra bit of annotations, the API can be used to process the set of annotations at compile time to generate the EFBO-compatible storyboard facts in OWL. Once we have the annotations, the annotated source code artifacts can be synchronized with the EFBO ontology. The future changes of functionalities based on the source code can then have a detectable model for the EFBO ontology.

Figure 6 provides a high-level depiction of the EFBO-based functional consistency validation system along with the set of core steps involved within the system. Just like the instantiated functionality model for the example login interface, we can have a set of additional login interface models instantiated for the EFBO ontology. Once instantiated, we can validate different kinds of functional consistencies between the source models. The mapping between the instance entities will be constrained through the automated classification on the EFBO ontology. In other words, the inferred classification of the ontology will have a clear indication of the mappable types of instances between two sources. For example, all the instances of agents will be classified as the type of *Agent* for each of the sources. Similarly, all the actions performed by each of the agents will be classified as the type of *Action*. We can use the *owl:sameAs* property between a pair of mappable instances in order to declare the individual pairs to be treated as identical for the ontology reasoner. Based on the set of reasoning categories in Section 5.1, we can then validate the level of functional consistencies between the source models. This kind of consistency validation can be quite useful for the cross-platform applications that are developed and evolved independently with the same set of functional goals.

6. Conclusion

The contribution of this paper can be summarized as a systematization of the functional behavior modelling for the event-based systems. We have presented a representation facility called the EFBO that can effectively specify the functional entities of a system in such a way so that their existence in the system can be thoroughly reasoned in a rigorous, conceptual manner. We have provided an example of the EFBO-based functional behavior modelling in order to demonstrate the potentials of our representational approach. We have highlighted a number of useful functional reasoning categories and use case scenarios where the EFBO-based representation can become quite effective.

Based on our experience, while there exist numerous ontologies for various application domains, when it comes to actually modelling the real-life systems, only a handful of them can be considered as somewhat useful. While the ontologies are not meant to incorporate instances within their formal knowledge structures, an ontology that lacks the mechanisms to maximally instantiate the real-life data can be quite frustrating. Even with the advancements of various automated reasoning supports, most of the ontologies, in this latter regard, exhibit the analogous limitations of the formal method based approaches as mentioned in Section 1. Compared to most of the existing approaches of representing functional behaviors, one of the distinguishing features of our approach is that we have considered the notion of Commonsense Reasoning to be the core basis to handle the functional reasoning questions. We have carefully engineered the EFBO ontology with a minimal set of required classes and properties so that the ontology can serve the specific purpose of functional reasoning on the event-based system. As we wanted to have a natural way to describe the functional behavior of a system, we have incorporated a number of useful properties and their inverses within the EFBO ontology which provides a flexible linguistic support to describe the behavior of an intended system. The EFBO-based instantiation process is therefore quite intuitive and flexible. This was one of our design choices in order to achieve the maximum usability of the ontology without compromising the logical rigour that we needed to achieve an effective functional reasoning mechanism.

For any artificial systems of knowledge, a set of functional goals can often be achieved in multiple different ways on different environments. However, how to reason about the multiplicities of ways that leads to the same set of functional achievements, or vice-versa, is mostly dependent on our cognitive processing abilities. This article has the potential to address some of the key principles that can support the usual cognitive practice, and ultimately, enhance our reasoning abilities in a systematic, more efficient, machine-assisted fashion. Our research will contribute to this latter aspect of reasoning about the world, which is one of the almost never-ending quests of commonsense reasoning in Artificial Intelligence.

Acknowledgement

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Sunitha Abburu. A survey on ontology reasoners and comparison. *International Journal of Computer Applications*, 57, 2012.
2. Ernest Davis and Gary Marcus. Commonsense reasoning and commonsense knowledge in artificial intelligence. *Communications of the ACM*, 58(9):92–103, 2015.
3. Behrouz Homayoun Far and A Halim Elamy. Functional reasoning theories: Problems and perspectives. *AIE EDAM*, 19(02):75–88, 2005.
4. Pierre Grenon and Barry Smith. Snap and span: Towards dynamic spatial ontology. *Spatial cognition and computation*, 4(1):69–104, 2004.
5. Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.
6. Yilan Gu and Mikhail Soutchanski. A description logic based situation calculus. *Annals of Mathematics and AI*, 58(1-2):3–83, 2010.
7. David M Hilbert and David F Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys (CSUR)*, 32(4):384–421, 2000.
8. Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.
9. Erik T Mueller. Automating commonsense reasoning using the event calculus. *Comm. of the ACM*, 52(1):113–117, 2009.
10. Kathy Panton, Cynthia Matuszek, Douglas Lenat, Dave Schneider, Michael Witbrock, Nick Siegel, and Blake Shepard. Common sense reasoning–from cyc to intelligent assistant. In *Ambient Intelligence in Everyday Life*, pages 1–31. Springer, 2006.
11. Murray Shanahan. The event calculus explained. In *Artificial intelligence today*, pages 409–430. Springer, 1999.
12. Xiao Hang Wang, Da Qing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using OWL. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 18–22. IEEE, 2004.