Scenario Testing on UML Class Diagrams using Description Logic

Hendrina F. Harmse, Katarina Britz, and Aurona Gerber

CSIR Meraka Institute and UKZN, South Africa

Abstract. As part of the requirements phase of building a software system, a conceptual model could be created which describes the information and processes of the business. This conceptual model is often expressed as a UML class diagram. In an attempt to validate the UML class diagram, stakeholders test various usage scenarios against it. The availablity of formal reasoning procedures will greatly ease finding associated inconsistencies. To date, substantial research has been done on transforming UML class diagrams into different Description Logics. In this paper we explore how Description Logics can be used to fascilitate scenario testing as a means to validate the UML class diagrams. Since identity constraints on UML class diagrams have not been mapped to a Description Logic as yet, we evaluate the suitability of using either $\mathcal{DLR}_{ifd}$ or $\mathcal{SROIQ}(\mathcal{D})$ for this purpose.

1 Introduction

Stakeholders often find it difficult to articulate the requirements of a system. In order to facilitate the requirements elicitation process, stakeholders are encouraged to think through specific scenarios. Scenarios are sequences of results that the intended system is expected to produce [5][9]. A conceptual model is developed from the elicited requirements. Scenarios are often used to test and validate this model to ensure that it does represent the requirements.

Depending on the complexity of the requirements, even testing only a subset of scenarios can still be error prone. Availability of formal reasoning procedures can help elucidate consistencies and/or inconsistencies. As mentioned, UML class diagrams are often used to represent the conceptual model, and a formal means to describe such models is therefore desirable. Indeed, several means exist for describing UML class diagrams formally. In this paper we focus our attention on Description Logics for this purpose.

Description Logics are decidable fragments of first order logic that are specifically designed for the conceptual representation of an application domain in terms of classes and relationships between classes [2]. In particular, Berardi, et al. [2] have done work to describe UML class diagrams in the Description Logic $\mathcal{DLR}_{ifd}$ and $\mathcal{ALCQI}$. Recent research described UML class diagrams in OWL 2 [11] which is based on the Description Logic $\mathcal{SROIQ}(\mathcal{D})$ [4].

Our research differs from previous efforts to describe UML class diagrams in Description Logics in the following important ways. Firstly, our aim is to limit the scope of UML class diagrams to a subset of the business domain. This approach supports the fact that stakeholders often only know a subset of the business domain. The added benefit is that, since the size of the input is limited, concerns of computational complexity are reduced. Secondly, research to date has focused on reasoning over UML class diagrams without objects. Our approach is to specifically include objects in UML class diagrams in order to represent a specific scenario. This approach is closely aligned with stakeholders giving consideration to a particular scenario or a group of scenarios [5].

The rest of the paper is organized as follows. In Section 2 we provide the motivation and contextualization for testing consistency/inconsistency of sub domains of the business via scenario testing rather than the conceptual model of the complete business domain as part of the requirements phase. We provide an illustrating example of a sub domain from the Medical Aid industry in Section 3. In Section 4 we explain different scenario testing approaches on hand of the example provided in Section 3. We additionally discuss the benefits and challenges of representing scenario tests using UML class diagrams and Description Logics. In Section 5 we explore the possibility of mapping identity constraints on UML class diagrams to either $\mathcal{DLR}_{ifd}$ or $\mathcal{SROIQ}(\mathcal{D})$. We also touch on a limitation of UML identity constraints. Based on our findings we suggest possible research contributions and the relevant next steps in Section 6.

2 The Case for Scenario Testing of Sub Domains

In this section we explain the value of testing a sub domain rather than the complete domain of a business (2.1). We provide the benefits and limitations of scenario testing (2.2) and we contextualize the use of scenario testing (2.3).

2.1 Testing Subdomain versus Complete Domain

From a business perspective there is substantial motivation for considering a sub domain of the business rather than the complete business domain. A single business stakeholder often only have knowledge of the portion of the business they are exposed to [5][9]. Giving consideration only to a particular sub domain allows business stakeholders to focus only on the concerns that they are in control of and which are of importance to them.

Not all business functions have the same risk profile. Therefore spending equal amounts of effort on all business functions is not cost effective. In the industry this fact has been addressed for system implementation testing by quality assurance experts through the use of risk based testing. Risk based testing aims to prioritize and allocate testing resources according to the risk profile of the system under test [6]. Applying a similar approach to the formalization of the business can result in some parts of the business being formalized while other parts are not formalized. This flexibility gives business stakeholders the power to decide where effort needs to be spent most.

The fact that this is not an all-or-nothing approach will facilitate industry adoption. This allows business to try out the idea in the small without making any large scale financial commitments. This will enable business to have a quick return on investment (ROI).

During systems development only a subset of the business domain is considered. Newly developed systems are often developed using an iterative rapid development methodology rather than a waterfall methodology. With iterative methodologies only a subset of the system is considered for development at any given time. When existing systems are enhanced, the scope of the enhancement is usually limited in an attempt to limit the business risk. Thus for both newly developed systems and enhancements to existing systems it is common practice in the industry to consider only a subset of the business domain [1].

2.2 Benefits and Limitations of Scenario Testing

Using scenarios to test the consistency/inconsistency of a sub domain of the business has a number of advantages.

Capturing requirements via scenarios, and validating the captured requirements through scenario testing, are practices that are well established within the industry. If formal reasoning procedures can be applied to scenario testing, it will reduce the margin of error even further while being closely aligned with how practitioners work. This will reduce the learning curve of adopting our approach.

Limiting the checking of UML class diagrams to a scenario test, or group of scenario tests, allows for a better understanding of the nature of inconsistencies: Explaining why a specific scenario test fails is more understandable to stakeholders than explaining mathematical logic concepts like consistency/inconsistency.

A limitation of testing consistency/inconsistency of a sub domain via scenario testing is that it cannot ascertain that the complete business domain is consistent or find all inconsistencies present in the complete business domain. However, from a business perspective, this limitation represents a reasonable compromise since it is seldom cost effective to formalize the complete business domain.

2.3 Scenario Testing Contextualization

Our aim with a Description Logic formalization of scenario testing is to strengthen existing scenario testing and software testing approaches. Existing scenario testing approaches will be used to determine the scenario tests that will need to be formalized while traditional software testing approaches will still be used to verify the implementation of the business specification.

In some cases business may decide that implementing a specific business function in a software system is not cost effective. Even for such niche cases, a Description Logics based approach to scenario testing will still be valuable and viable. It can still indicate consistencies and/or inconsistencies, while not being dependent on an implemented software system.

In this paper we explain how scenario tests can be defined using UML class diagrams. The use of UML class diagrams is aimed at fascilitating industry adoption. It is completely viable to define scenario tests directly in Description Logics.

3 Example Business Domain

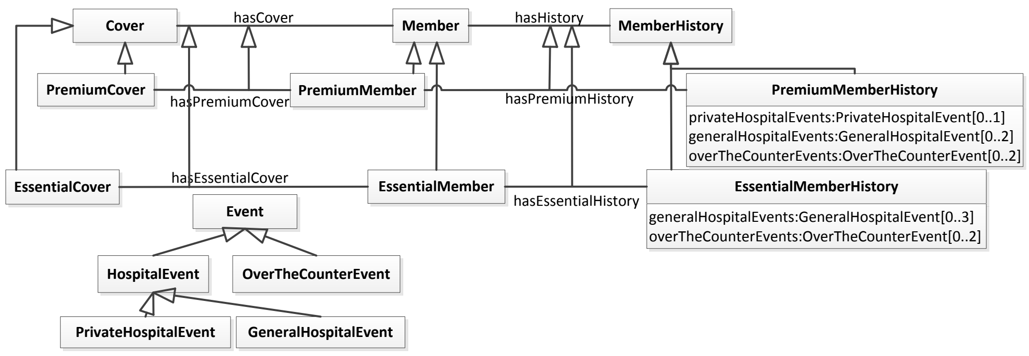

Fig. 1. A UML Class diagram modeling a sub domain of a Medical Aid provider

In Fig. 1 we provide an example UML class diagram that models a sub domain of a Medical Aid provider. The Medical Aid provider has a number of business rules. A member must have an associated Cover. Two types of cover are provided namely Premium Cover and Essential Cover. Each type of Cover will pay for certain medical Events based on some predefined rules. In this example the only Events considered are when a person visits the hospital or buys medicine without a prescription, i.e. over the counter medicine. For hospital visits a person may choose to visit a more expensive (Private) or less expensive (General) hospital. Premium Cover allows up to 1 visit to a Private Hospital, up to 2 visits to a General Hospital and medicine may be bought up to 2 times without a prescription. Essential Cover differs from Premium Cover in that it will not pay for any visits to a Private Hospital, while it will pay for up to 3 visits to a General Hospital.

In order to keep Fig. 1 terse, we made use of UML defaults where possible. The particular defaults that are applicable to Fig. 1 relate to generalizations and multiplicities on associations. When generalizations are not annotated, the default annotation {disjoint, complete} is assumed. “Disjoint” indicates that the subclasses of the generalization have no object instances in common. “Complete” indicates that when an object instance is an instance of the general class, it is also an instance of at least one subclass. Absence of multiplicities on an association defaults to the multiplicity [1..1] [7].

For the most part we made use of UML class diagram features as explained in Berardi, et al. [2]. The only exception is that we made use of association specialization for example between the hasEssentialCover and hasPremiumCover associations and the hasCover association. hasCover represents a more general association than hasEssentialCover and hasPremiumCover.

Finally, it may seem peculiar that there are no explicit associations between the Event classes and any other classes in the model. However, associations between the Event classes and MemberHistory classes are implied based on the attributes of the PremiumMemberHistory and EssentialMemberHistory classes. Adding an attribute to a class is equivalent to adding an association [7]. Thus, while adding these associations are admissible, they are redundant.

We provide the $\mathcal{SROIQ}(\mathcal{D})$ representation of the UML class diagram of Fig. 1 in the Appendix. All the constructs in Fig. 1 translate to $\mathcal{SROIQ}(\mathcal{D})$ without loss of information, and in line with the intended UML semantics [2]. Although this example does not require the use of $\mathcal{SROIQ}(\mathcal{D})$ constructs such as nominals or concrete domains, these have not been excluded intentionally, and can feature in other scenarios.

4 Scenario Testing

In this section we provide more detail about our proposed scenario testing approach. We start with identifying three approaches to scenario testing in 4.1. Modelling scenarios via either UML class diagrams or Description Logics yield some unexpected challenges, which we discuss in 4.2. Challenges specific to the use of UML class diagrams are discussed in 4.3, while the benefits of using Description Logics for scenario testing are discussed in 4.4.

4.1 Approaches

We identify three approaches to scenario testing. Firstly, a scenario that is expected to be successful can be tested for consistency. Secondly, a specific scenario may be expected to be unsuccesful and hence can be tested for inconsistency. Lastly, classification can be applied to object instances to ascertain that the applied business rules have the desired effect. We discuss each of the three approaches on hand of the Medical Aid provider example introduced in Section 3.

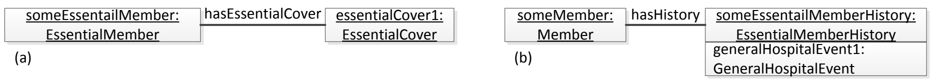
someEssentailMember: EssentialMember
hasEssentialCover
essentialCover1: EssentialCover
(a)
someMember: Member
hasHistory
someEssentailMemberHistory: EssentialMemberHistory
generalHospitalEvent1: GeneralHospitalEvent
(b)

Fig. 2. Object instances defining successful scenarios

We give consideration to successful scenarios in Fig. 2. In Fig. 2(a) we want to ensure that an Essential Member can have Essential Cover. In Fig. 2(b), since both Premium Cover and Essential Cover allows a member to visit a General Hospital, a member, irrespective of whether the member has Premium or Essential Cover, should be allowed to visit a General Hospital. Thus both the scenarios illustrated in Fig. 2 represent successful scenarios. The respective Description Logic respresentations for the scenarios in Fig. 2 are given in Examples 1 and 2.

Example 1.
`EssentialCover(essentialCover1)`
`EssentialMember(someEssentialMember)`
`hasEssentialCover(someEssentialMember, essentialCover1)`

Example 2.
`GeneralHospitalEvent(generalHospitalEvent1)`
`Member(someMember)`
`EssentialMemberHistory(essentialMemberHistory1)`
`hasHistory(someMember, essentialMemberHistory1)`
`generalHospitalEvents(essentialMemberHistory1, generalHospitalEvent1)`

Business is also interested in scenarios that should be unsuccessful for their given business context. Being able to test for unsuccesful scenarios are of value in ensuring that the business rules are defined sufficiently to disallow such scenarios. As an example from our Medical Aid provider an unsuccesful scenario will arise if any Member happens to be registered for two different Cover options (see Fig. 3(a) and Example 3). Also we will not expect an Essential Member to be allowed to visit a Private Hospital (see Fig. 3(b) and Example 5). Note that in Description Logic, due to the open world assumption, we need to explicitly state that cover1 and cover2 are different individuals (Example 3).

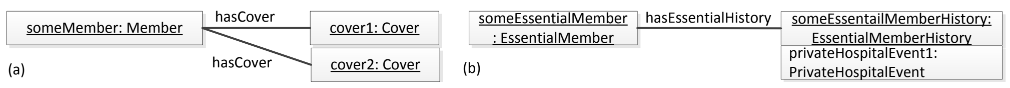
someMember: Member
hasCover
cover1: Cover
hasCover
cover2: Cover
(a)
someEssentialMember : EssentialMember
hasEssentialHistory
someEssentailMemberHistory: EssentialMemberHistory
privateHospitalEvent1: PrivateHospitalEvent
(b)

Fig. 3. Object instances defining unsuccessful scenarios

Example 3.
`Cover(cover1)`
`Cover(cover2)`
`Member(someMember)`
`hasCover(someMember, cover1)`
`hasCover(someMember, cover2)`
`cover1` $\not\approx$ `cover2`

Example 4.
`EssentialMemberHistory(essentialMemberHistory1)`
`PrivateHospitalEvent(privateHospitalEvent1)`
`EssentialMember(someEssentialMember)`
`privateHospitalEvents(essentialMemberHistory1, privateHospitalEvent1)`
`hasHistory(someEssentialMember, essentialMemberHistory1)`

In order to discern how different business rules will classify instances, it will be useful to apply classification to object instances to validate that the applied business rules have the desired effect. In Fig. 4 the aim is to check whether application of the appropriate business rules will result in the expected classification. As an example we will expect for the Medical Aid provider that any object instance that has an associated Cover should be a Member. Also, any object instance that has an associated Essential Cover should be an Essential Member. This is useful to ensure that the scope of business rules are correct. We will for instance not want that any object instance that has an associated Premium Cover to be considered an Essential Member. See Example 5 for the Description Logic respresentation of Fig. 4 along with related inferences.

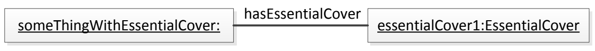

Fig. 4. Object instance with no type information

Example 5.
$K \cup \{\texttt{EssentialCover}(\texttt{essentialCover1}),$
$\texttt{hasEssentialCover}(\texttt{someThingWithEssentialCover}, \texttt{essentialCover1})\}$
$\models \texttt{EssentialMember}(\texttt{someThingWithEssentialCover})$

$K \cup \{\texttt{EssentialCover}(\texttt{essentialCover1}),$
$\texttt{hasEssentialCover}(\texttt{someThingWithEssentialCover}, \texttt{essentialCover1})\}$
$\models \texttt{Member}(\texttt{someThingWithEssentialCover})$

4.2 Challenges of Modelling Scenario Tests

One of the challenges we encountered in modelling scenario tests is the need to determine how to model business rules such that they can be validated through scenario tests. In our Medical Aid provider example (see Fig. 1), one will intuitively expect the rules for Premium Cover to be defined on the PremiumCover class and concept repectively for UML class diagrams and Description Logics. This is however not the case.

Indeed, we found it best to make a clear distinction between classes (resp. concepts) that determine applicable rules and classes (resp. concepts) that are responsible for applying the rules. In particular we found it best to model the business rules on the class (resp. concept) where the rules are applied. An association (resp. role) between the applying class (resp. concept) and determining class (resp. concept) is responsible for indicating which rules need to be applied.

For instance, in our Medical Aid provider example, the PremiumMember-History class is where the rules are applied while hasPremiumHistory is an association to the PremiumMember class which is used to determine the rules that are applicable.

4.3 UML Class Diagram Challenges

The advantage of using UML class diagrams for the definition of scenario tests is that it leverages tools that are well established in the industry. The disadvantage is that expressing business rules via UML class diagrams is not intuitive. It is for example difficult to express negative information and conditional logic using UML class diagrams. Both can be simulated to some extent using generalizations, but the simulation is not intuitive.

In our Medical Aid provider example (see Fig. 1) forms of conditional logic and negative information are applied through for instance the PremiumMemberHistory and EssentialMemberHistory classes. Conditional logic is applied in that only if a member is a Premium Member are they allowed to visit a Private Hospital. The fact that an Essential Member is not allowed to visit a Private Hospital (a form of negative information), is implied through the fact that the PremiumMemberHistory and EssentialMemberHistory classes are complete and disjoint generalizations of MemberHistory.

4.4 Benefits of using Description Logics for Scenario Testing

There are a number of benefits to using a Description Logic formalization of scenario testing. Firstly, scenario testing approaches rely on traditional testing approaches which have no formal reasoning support. Therefore traditional testing approaches can at most identify problems but they cannot show their absence. Description Logics, however, are equiped with formal reasoning procedures.

In our scenario testing approaches we used consistency checking and classification reasoning procedures for verifying successful/unsuccessful and classification scenario tests. Based on the reasoning procedures used, it is possible to provide proofs of why a scenario test is consistent or inconsistent or why an object instance has been classified in a specific way. This is not possible with the traditional testing approaches.

Secondly, logical entailment can be used to gain deeper insight into the implicit consequences of the model. This is especially useful when unexpected results need to be investigated. In the absence of logical entailment, stakeholders can only guess at the reasons for the unexpected results.

Thirdly, due to the mathematical basis of Description Logics, they do not suffer of the ambiguities that plagues the UML specification. Indeed, Description Logics of UML class diagrams can be used to make the intended meaning of a UML class diagram explicit.

Fourthly, using Description Logics it is possible to validate the business specification before the business specification is implemented in software. Addressing defects during the requirements phase is more cost effective than addressing it later [3].

Lastly, it is possible to use Description Logics directly to implement scenario testing with. In specific using Description Logics directly avoids the challenges mentioned regarding the use of UML class diagrams (4.3).

5 UML Class Diagram Identity Constraints

Identity constraints on UML class diagrams have not been translated to Description Logics as yet. In this section we explore means of addressing identity constraints in the context of scenario testing. In 5.1 we discuss identity constraints on UML class diagrams as specified in the UML specification [7] and we highlight a gap in specification w.r.t. identity constraints. We discuss the challenges of translating identity constraints on UML class diagrams in 5.2 and we show that identity constraints, as implemented in $\mathcal{SROIQ}(\mathcal{D})$, is viable in the context of scenario testing.

5.1 UML Identity Constraints Discussion

The UML specification allows one to indicate that an attribute or attributes can be used to identify objects of a class uniquely via an id property modifier [7]. In Fig. 5 (a) and (b) we respectively show a member identified by a combination of his/her name, surname and date of birth and a member identified by his/her Medical Aid number.

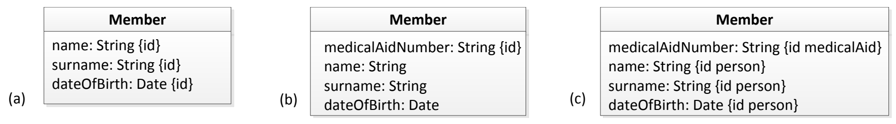

Fig. 5. Examples of the id property modifier

Interestingly the UML specification does not make provision for specifying multiple identity constraints on a single class. In the case of Fig. 5(b) it may be convenient to be able identify members by either Medical Aid number or name, surname and date of birth. In Fig. 5 (c) we propose the introduction of an alias to make it possible to identify multiple identity constraints on a single class.

5.2 Identity Constraint Challenges

The challenge of mapping UML identity constraints to Description Logics is that identity constraints in UML are usually applied to datatypes. It is a well-known fact that identity contraints in the presence of datatypes leads to undecidability even for the Description Logic $\mathcal{ALC}$ extended accordingly [8][10]. $\mathcal{DLR}_{ifd}$ includes support for identity constraints on concepts, but not for identity constraints on datatypes. As such $\mathcal{DLR}_{ifd}$ is not well suited for representing identity constraints on UML class diagrams.

For the Description Logic $\mathcal{SROIQ}(\mathcal{D})$, Parsia, et al. [10] provides a reasonable compromise by allowing reasoning on key constraints on named individuals rather than generated individuals. Object instances in UML map to named individuals in Description Logics. Therefore the identity constraint approach of Parsia, et al. [10] provides a suitable compromise for scenario testing. Furthermore, with the identity constraint approach of Parsia, et al. [10] it is possible to define multiple identity constraints on a concept. This ability addresses the gap identified in the UML specification in 5.1.

6 Further Research and Conclusion

From our findings discussed in this paper, there is a clear case to be made for Description Logic based scenario testing and further research in this regard is justified. We conclude by briefly mentioning areas for further investigation.

A challenge that still needs to be addressed, as discussed in 4.2, is that modelling for scenario testing may not be obvious. Therefore providing clear guidelines as to how to model various business rules in the context of scenario testing will be of value to practitioners.

The fact the scenario testing is based on object instances rather than classes alone, presents additional opportunities for representing UML class diagram features in Description Logics, as we have shown in our discussion on identity constraints in Section 5. It will be interesting to explore what other UML class diagram features can benefit similarly.

Lastly, it will make sense to explore means through which scenario testing can benefit from the query answering capabilities of Description Logics.

References

1. David Avison and Guy Fitzgerald, *Information Systems Development: Methodologies, Techniques and Tools*, 4 ed., McGraw-Hill Higher Education, 2006.
2. Daniela Berardi, Andrea Cali, Diego Calvanese, and Giuseppe Di Giacomo, *Reasoning on UML Class Diagrams*, ARTIFICIAL INTELLIGENCE (2003), 2005.
3. B. Boehm and V.R. Basili, *Software defect reduction top 10 list*, Foundations of empirical software engineering: the legacy of Victor R. Basili (2005), 426.
4. B. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patelschneider, and U. Sattler, *OWL 2: The next step for OWL*, Web Semantics: Science, Services and Agents on the World Wide Web **6** (2008), no. 4, 309–322.
5. M. Elizabeth C. Hull, Ken Jackson, and Jeremy Dick (eds.), *Requirements Engineering, Third Edition.*, Springer, 2011.
6. M.L. Hutcheson, *Software testing fundamentals: methods and metrics*, Wiley, 2003.
7. ISO, *Information Technology - Object Management Group Unified Modeling Language (OMG UML), Superstructure*, April 2012.
8. Carsten Lutz, Carlos Areces, Ian Horrocks, and Ulrike Sattler, *Keys, Nominals, and Concrete Domains.*, J. Artif. Intell. Res. (JAIR) **23** (2005), 667–726.
9. B. Nuseibeh and S. Easterbrook, *Requirements engineering: a roadmap*, Proceedings of the Conference on the Future of Software Engineering, ACM, 2000, pp. 35–46.
10. Bijan Parsia, Ulrike Sattler, and Thomas Schneider 0002, *Easy Keys for OWL.*, OWLED (Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, eds.), CEUR Workshop Proceedings, vol. 432, CEUR-WS.org, 2008.
11. Jesper Zedlitz, Jan Jörke, and Norbert Luttenberger, *From UML to OWL 2*, Knowledge Technology (Dickson Lukose, AbdulRahim Ahmad, and Azizah Suliman, eds.), Communications in Computer and Information Science, vol. 295, Springer Berlin Heidelberg, 2012, pp. 154–163.

Appendix

$\texttt{PremiumCover} \sqsubseteq \texttt{Cover}$
$\texttt{EssentialCover} \sqsubseteq \texttt{Cover}$
$\texttt{PremiumMember} \sqsubseteq \texttt{Member}$
$\texttt{EssentialMember} \sqsubseteq \texttt{Member}$

$\top \sqsubseteq \forall\texttt{hasCover}.\texttt{Cover} \sqcap \forall\texttt{hasCover}^{-}.\texttt{Member}$
$\top \sqsubseteq (\exists\texttt{hasCover}.\top) \sqcap (\leq 1\,\texttt{hasCover}.\top)$
$\top \sqsubseteq (\exists\texttt{hasCover}^{-}.\top) \sqcap (\leq 1\,\texttt{hasCover}^{-}.\top)$

$\top \sqsubseteq \forall\texttt{hasPremiumCover}.\texttt{PremiumCover} \sqcap \forall\texttt{hasPremiumCover}^{-}.\texttt{PremiumMember}$
$\top \sqsubseteq (\exists\texttt{hasPremiumCover}.\top) \sqcap (\leq 1\,\texttt{hasPremiumCover}.\top)$
$\top \sqsubseteq (\exists\texttt{hasPremiumCover}^{-}.\top) \sqcap (\leq 1\,\texttt{hasPremiumCover}^{-}.\top)$

$\top \sqsubseteq \forall\texttt{hasEssentialCover}.\texttt{EssentialCover}\sqcap\forall\texttt{hasEssentialCover}^{-}.\texttt{EssentialMember}$
$\top \sqsubseteq (\exists\texttt{hasEssentialCover}.\top) \sqcap (\leq 1\,\texttt{hasEssentialCover}.\top)$
$\top \sqsubseteq (\exists\texttt{hasEssentialCover}^{-}.\top) \sqcap (\leq 1\,\texttt{hasEssentialCover}^{-}.\top)$

$\texttt{hasPremiumCover} \sqsubseteq \texttt{hasCover}$
$\texttt{hasEssentialCover} \sqsubseteq \texttt{hasCover}$

$\texttt{HospitalEvent} \sqsubseteq \texttt{Event}$
$\texttt{PrivateHospitalEvent} \sqsubseteq \texttt{HospitalEvent}$
$\texttt{GeneralHospitalEvent} \sqsubseteq \texttt{HospitalEvent}$
$\texttt{OverTheCounterEvent} \sqsubseteq \texttt{Event}$

$\texttt{PremiumMemberHistory} \sqsubseteq \texttt{MemberHistory}$
$\texttt{EssentialMemberHistory} \sqsubseteq \texttt{MemberHistory}$

$\top \sqsubseteq \forall\texttt{hasHistory}.\texttt{History} \sqcap \forall\texttt{hasHistory}^{-}.\texttt{Member}$
$\top \sqsubseteq (\exists\texttt{hasHistory}.\top) \sqcap (\leq 1\,\texttt{hasHistory}.\top)$
$\top \sqsubseteq (\exists\texttt{hasHistory}^{-}.\top) \sqcap (\leq 1\,\texttt{hasHistory}^{-}.\top)$

$\top \sqsubseteq \forall\texttt{hasPremiumHistory}.\texttt{PremiumHistory}\sqcap\forall\texttt{hasPremiumHistory}^{-}.\texttt{PremiumMember}$
$\top \sqsubseteq (\exists\texttt{hasPremiumHistory}.\top) \sqcap (\leq 1\,\texttt{hasPremiumHistory}.\top)$
$\top \sqsubseteq (\exists\texttt{hasPremiumHistory}^{-}.\top) \sqcap (\leq 1\,\texttt{hasPremiumHistory}^{-}.\top)$

$\top \sqsubseteq \forall\texttt{hasEssentialHistory}.\texttt{EssentialHistory}\sqcap\forall\texttt{hasEssentialHistory}^{-}.\texttt{EssentialMember}$
$\top \sqsubseteq (\exists\texttt{hasEssentialHistory}.\top) \sqcap (\leq 1\,\texttt{hasEssentialHistory}.\top)$
$\top \sqsubseteq (\exists\texttt{hasEssentialHistory}^{-}.\top) \sqcap (\leq 1\,\texttt{hasEssentialHistory}^{-}.\top)$

$\texttt{hasPremiumHistory} \sqsubseteq \texttt{hasHistory}$
$\texttt{hasEssentialHistory} \sqsubseteq \texttt{hasHistory}$

$$\texttt{PremiumMemberHistory} \sqsubseteq \forall \texttt{privateHospitalEvents}.\texttt{PrivateHospitalEvent}$$
$$\texttt{PremiumMemberHistory} \sqsubseteq (\leq 1\, \texttt{privateHospitalEvents}.\top)$$

$$\texttt{PremiumMemberHistory} \sqsubseteq \forall \texttt{generalHospitalEvents}.\texttt{GeneralHospitalEvent}$$
$$\texttt{PremiumMemberHistory} \sqsubseteq (\leq 2\, \texttt{generalHospitalEvents}.\top)$$

$$\texttt{PremiumMemberHistory} \sqsubseteq \forall \texttt{overTheCounterEvents}.\texttt{OverTheCounterEvent}$$
$$\texttt{PremiumMemberHistory} \sqsubseteq (\leq 2\, \texttt{overTheCounterEvents}.\top)$$

$$\texttt{EssentialMemberHistory} \sqsubseteq \forall \texttt{generalHospitalEvents}.\texttt{GeneralHospitalEvent}$$
$$\texttt{EssentialMemberHistory} \sqsubseteq (\leq 3\, \texttt{generalHospitalEvents}.\top)$$

$$\texttt{EssentialMemberHistory} \sqsubseteq \forall \texttt{overTheCounterEvents}.\texttt{OverTheCounterEvent}$$
$$\texttt{EssentialMemberHistory} \sqsubseteq (\leq 2\, \texttt{overTheCounterEvents}.\top)$$