IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Ingénierie des Systèmes d'Information
Vol. 28, No. 6, December, 2023, pp. 1533-1547
Journal homepage: http://iieta.org/journals/isi

Multi-Layer Consistency Validation of IoT Systems with UML Inheritance Dynamic Diagrams via SPIN Model Checking

Nabil Messaoudi[1*], Haouassi Hicham[1], Maarouk Toufik Messaoud[1], Hamdane Mohamed Elkamel[2]

[1] ICOSI Laboratory, Abbes Laghrour University, Khenchela 40000, Algeria
[2] Teachers' Training School of Constantine, MISC Laboratory, Abdelhamid Mehri University, Constantine 25000, Algeria

Corresponding Author Email: messaoudi.nabil@univ-khenchela.dz

https://doi.org/10.18280/isi.280610

Received: 6 October 2023
Revised: 27 November 2023
Accepted: 7 December 2023
Available online: 23 December 2023

Keywords:
Büchi automata, IoT healthcare, IoT, inheritances, SPIN Model Checker, UML dynamic diagram, multi-layer UML consistency checking, model transformations

ABSTRACT

The integration of the Unified Modeling Language (UML) with the Internet of Things (IoT) facilitates the multi-faceted modeling of complex IoT systems. Despite existing methodologies addressing UML coherence, the literature reveals a paucity of strategies for ensuring consistency between use cases and their manifestations in activity and sequence diagrams, particularly when inheritance is employed. This study delves into the validation of UML behavioral views, focusing on the coherence of use cases, activity diagrams, and sequence diagrams within IoT specifications through a multi-layered consistency approach. A methodology is presented for transforming IoT system specifications into Büchi automata, enabling consistency verification through the SPIN Model Checker. The robustness of this method is demonstrated through a case study involving a Healthcare IoT system, highlighting the utility of the proposed validation technique.

1. INTRODUCTION

The complexity inherent in the creation of sophisticated systems can be mitigated through the development of system models. Modeling serves as a critical instrument throughout the various stages of design, offering a lens through which the intricate nature of systems can be understood and crafted. Among the multitude of modeling languages, the Unified Modeling Language (UML) [1] stands out as a prevalent choice. UML enables the detailed representation of system structures and behaviors and has gained recognition as a de facto industry standard. The synergy between UML and the Internet of Things (IoT) proves particularly advantageous, allowing the precise definition of objects (e.g., devices, sensors), their attributes, and interactions, thus enhancing the clarity and complexity of IoT system development. Moreover, UML's capacity to visualize and manage complex device interactions is indispensable in IoT contexts [2, 3].

The objective of employing various models is to forge a more accurate representation of systems, which in turn facilitates code generation. The primary impetus for these models is the clear advantage of identifying and rectifying design flaws prior to the implementation of actual software components, as inconsistencies among software models often emerge as a significant challenge [4, 5].

In the realm of UML, inheritance extends beyond class diagrams to encompass use cases, enabling the inheritance of actors and use cases alike. The implications of simple and multiple inheritances warrant thorough examination. Despite this, current treatment of inheritance in Object-Oriented Design (OOD) typically restricts itself to static aspects, leaving dynamic behavior inheritance less defined. Consequently, when one use case inherits from another, there is an implicit extension of the activity and sequence diagrams associated with it [6, 7].

The ambiguity of UML semantics also contributes to the need for enhanced consistency, as multiple, sometimes conflicting, interpretations can be ascribed to a single UML expression. The verification of consistency traditionally involves rule definition and translation into formal languages. Consistency is achieved when distinct software models, which overlap in their description of the system's features, satisfy established conditions [8-10].

Management of model consistency typically encompasses three core tasks [5, 10, 11]: (i) the definition of consistency, (ii) the detection of inconsistencies, and (iii) the resolution thereof. Adherence to specified consistency rules, known as Rules Well Formedness (RWF), is essential for different UML diagrams (refer to the UML superstructure specification).

Model checking stands as a robust methodology for system verification, offering a formal mechanism to affirm the correctness of concurrent systems characterized by finite states. This approach involves the system's examination based on intricate algorithms to ascertain whether the given specifications are met. Furthermore, model checking is poised to aid in addressing inconsistencies within UML software models. Here, a formalism called Split Automata—a variant of automata adept at correlating activity diagrams with SPIN [12] is utilized.

This research is propelled by the scarcity of studies addressing the inheritance issue in use cases (UC), particularly the impact on the relationship between activity diagrams (AD)

and sequence diagrams (SD) [7]. A novel approach is proposed: a multi-layer and multi-level strategy for UML consistency checking. This includes horizontal consistency across generalizable diagrams and vertical consistency that leverages the relationship between generalizable and specialized diagrams. The challenge addressed is the assurance of conformity across UC, SD, and AD to analogous requirements. Figure 1 illustrates the primary activities involved.

The subsequent sections are structured as follows: Section 2 provides an overview of related works, Section 3 delineates the fundamental concepts within the researched domains, Sections 4 and 5 explicate the proposed approach including the consistency checking mechanism, Section 6 presents an IoT case study, and the paper culminates with a discussion on future research directions, accompanied by concluding remarks.

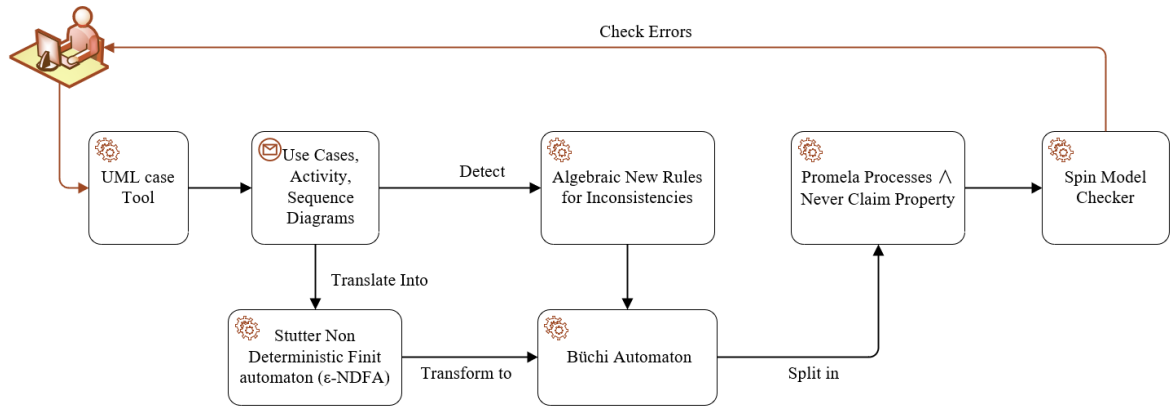

Figure 1. The process consistency between UC, AD, SD

2. RELATED WORKS

The delineation of UML consistency rules constitutes the focal point of the present study, situating itself within broader discourses pertaining to UML consistency. In this context, the contributions by the authors of studies [11, 13] are particularly instrumental, having conducted extensive research, compiled, and presented an array of UML diagram consistency rules.

In an organized compilation presented in Table 1 from [13], a total of 119 papers have been categorized based on a criterion of perspective. These studies encompass a spectrum of diagrams, encapsulating structural, behavioral, and integrated forms. Notably, out of the 119 papers surveyed, a mere 21 address behavioral consistency. It was further observed by the same author that in response to the inquiry regarding the application of UML consistency rules in software engineering activities, 253 instances pertained to verification purposes, whereas formalization accounted for only 14 instances. Among the 119 identified rules, a scant 5, equating to 4%, were proposed for inheritance, and these were presented in an informal manner. These rules predominantly concentrated on the consistency between class diagrams (DCs) and state machines (SMs).

Further scrutiny is given to the methodologies employed in 40 studies spanning from 2003 to 2016, detailed in Table 2, with a focus on both formal and informal techniques. A conspicuous majority of these studies have utilized formal methods, with logic and state transitions, which account for 73%, being particularly relevant to the scope of our investigation.

An innovative approach for the automatic verification of deadlocks and nondeterminism within UML activity diagrams is proposed [14]. This is achieved by introducing a compositional Communicating Sequential Processes (CSP) semantics for activity diagrams. This semantic framework facilitates the automatic derivation of CSP specifications from UML models, which are subsequently used as inputs for the FDR refinement checker, enabling the automated verification of deadlocks and nondeterminism.

The annexed papers referenced in this section pivot around use cases (UC), which serve as a basis for understanding and formalizing consistency. This research specifically refers to the papers designated as annexes to elucidate the mechanisms underpinning consistency.

In the domain of safety-critical systems, the articulation of safety requirements is a matter of paramount importance. An incremental approach to refining these requirements through scenarios is proposed by Chen et al. [15]. Once the consistency of the requirements is verified, these scenarios are subject to refinement, with their semantics being captured through a transformation into an intermediate semantic model conducive to formal verification. The validity of this transformation process is corroborated by aligning the requirements with actual execution traces.

The utility of the process algebra language LOTOS (Language of Temporal Ordering Specification) is examined in the study of Doostali et al. [16], where it serves as the chosen language for formal specification. Herein, a mapping methodology, designated USLP, is introduced to facilitate the translation of UML Statecharts into LOTOS processes. The fidelity of these mappings is substantiated through the establishment of an isomorphism between the Labelled Transition System (LTS) of a Statechart and that of the corresponding LOTOS specification.

A rule-based approach to UML consistency rules is explored by Kalibatiene et al. [17], where the authors have distilled 50 rules from a corpus of eight articles published between 2008 and 2011. Subsequent investigation within this research confirms the absence of rules pertaining to inheritance in the aforementioned set.

The need for a formal syntax and semantics for variations of Use Case Diagrams (UCD) is addressed in the study of Kautz et al. [18]. In response to the absence of a standardized language or formal semantics for UC representations, the

authors present a semantic differencing operator and subject it to empirical validation, highlighting the challenge of discerning relationships between similar diagrams.

Further extending the discussion on use case representation, Almendros-Jiménez and Iribarne [8] present a methodology that employs Sequence Diagrams (SD) to describe Use Case (UC) narratives, identifying inclusion and generalization relationships amongst UCs as a part of their approach.

In the study of Sapna et al. [19], the elements of use case, activity, and sequence diagrams are delineated using a schema table, albeit the focus is restricted to a subset of elements: use case, actor, activity, message, and object. The paper posits that each actor in a use case diagram corresponds to a class in an activity diagram and establishes a linkage between the objects and messages in a sequence diagram to a class and its methods in a class diagram. The authors introduce two consistency rules between use case and sequence diagrams, utilizing Object Constraint Language (OCL) for expression.

The formalization of consistency across various UML diagrams, including UC, AD, Sequence Diagrams (SD), and Statecharts (SC), is undertaken using coloured Petri nets in [5]. Consistency transitions are defined for use case, action, and execution occurrences. Notably, inheritance is not considered within their framework.

Table 1. Consistency from model view

	Model View		
	One View	**Multiview**	
Structural	44	00	44
behavioural	26	21	47
Struc&Beh	04	24	28
Total	74	45	119

Table 2. Consistency from formal view

Formal Techniques		**Non Formal Techniques**	
State.Transiti	13	OCL+Const.integrity	07
Logic	13	Sanity Consistency	01
Process.Alge	02	Xml rules+QVT	03
Total Formal	29	Total Non formal	11

Table 3. (a) Comparative related works

Study	**Year**	**FormTech.**	**Diagrams**
[5]	2006	CPN	UC, SD, AD
[9]	2011	--	UC, AD
[14]	2020	FDR, CSP	AD
[15]	2022	--	Doc.of UC, AD
[16]	2023	Lotos	UML stateCharts
[17]	2013	OCL	SM, SD
[18]	2022	Logical	UC
[19]	2007	OCL	UC, COD, AD, CD
This Paper	2023	SPIN, Log.	UC, AD, SD

Table 3. (b) Comparative related works

Study	**Year**	**(R, T)**	**S**	**M**
[5]	2006	T	--	--
[9]	2011	R	S	--
[14]	2020	T	--	--
[15]	2022	T	--	--
[16]	2023	T	--	--
[17]	2013	R	--	--
[18]	2022	T		
[19]	2007	R	--	--
This Paper	2023	R, T	√	√

This paper distinguishes itself from related works through a systematic comparison, encapsulated in Table 3. The comparison takes into account several criteria: the formal techniques employed (FormTech), the UML diagrams under consideration (Diagrams), and the presence or absence of simple (S) and multiple inheritance (M). Additionally, the paper evaluates the proposal of consistency rules (R) and the transformation of these rules into formal models (T). Articles not addressing a specific concept are indicated with '--'.

Building upon the foundation laid by previous studies, this paper adopts a logical approach to describe the elements of UC, AD, and SD, as well as their inter-consistencies. New consistency rules are proposed, refining and augmenting those previously established. These proposed standards are exemplified through a UML model incorporating the relevant diagrams. Moreover, the constituents of the consistency rules are delineated, with explicit justifications furnished for each component.

3. BACKGROUND

This section describes the fundamental notions of the domains that served as the foundation for our approach.

3.1 Use cases-oriented development

Table 4. IoT Use Case documentation

Use Case N°	**Use Case Description**
Goal in Context	Description of the use IoT case.
Scope & Level	Scope and level of the considered system.
Primary/Secondary IoT Actors	Role name or description of the primary and secondary actors for the use IoT case, sensors, actuators, cloud or other associated systems.
IoT Trigger	Which action of the primary/secondary actors initiate the IoT use case e.g., Available sensors.
Stakeholder &Interest	Name of the stakeholder and interest of the stakeholder in the IoT use case e.g SmartPhone.
Preconditions	Expected state of the system or its environment before the IoT use case may be applied eg. Data available from sensors.
Postconditions on success	Expected state of the system or its environment after successful completion of the IoT use case.
Postconditions on failure	Expected state of the system or its environment after unsuccessful completion of the IoT use case.
IoT UC Description	Flow of events that are normally performed in the use IoT case
Alternative Description	Flow of events that are performed in alternative scenarios (numbered).
Exceptions	Failure modes or deviations from the normal IoT use Case
IoT Use Case elationships	IoT use cases that are included, extended, or inherited.
Concurrent IoT UC	IoT use cases that can be applied concurrently to the current IoT use case.

The documentation for the use case is formatted in the form of a template. We take into account the pertinent facts that will be used as a foundation for the determination of some consistency standards. The preconditions, postconditions, and

explanation of the use case scenario's basic and alternative options are the most crucial entries to include. In what follows, we are going to focus mostly on the concepts of triggers, pre-conditions, and post-conditions to propose some rules in the context of inheritance.

The Table 4 provides an illustration of a use case template, with the various topic definitions being derived from [20-22] and adapted to the context of an IoT application. It is a representation of the typical components of an IoT use case that may be significant for conveying how a user interacts with a system.

3.2 Inheritance of UML behaviour

The term *simple inheritance* is used when a class only inherits from a single other class, while the term *multiple inheritance* is used when multiple classes contribute to the class's inheritance tree [23].

The inheritance relation is irreflexive, i.e., a class A cannot inherit from itself. The relation is transitive if class A inherits from class B and B inherits from class C; then A inherits from C. The relation is non-symmetric; if class A inherits from class B, then B does not inherit from A.

Also, as specified in the UML literature, an actor can inherit from another actor (single inheritance), and an actor can inherit from other actors (multiple inheritances). The exact definitions are also applicable to UC.

The four inheritance concepts referred to in the study of van der Aalst [6] will be discussed to analyse the effect of inheritance in the context of UC. Since the basic ideas are general, they can be applied to any IoT UML behaviour diagram.

The author presented two types of behavioural inheritances, (i) protocol inheritance and (ii) projection inheritance (see Figure 2).

Protocol inheritance means that if the external behaviour of p and q cannot be distinguished when only actions of p that are also present in q are executed, then p is a sub-behaviour of q; *which is conforms to blocking actions new in p and* any sequence of actions invocable on the super-behaviour can be invoked on the sub-behaviour.

Projection inheritance means that p is a sub-behaviour of q, if the external behaviour of p and q cannot be distinguished when arbitrary actions of p are performed, but can be distinguished when only the effects of actions that are also present in q are considered; *which conforms to hiding actions new in p and* any sequence of actions observable from the sub-behaviour should correspond to an observable sequence of the super-behaviour.

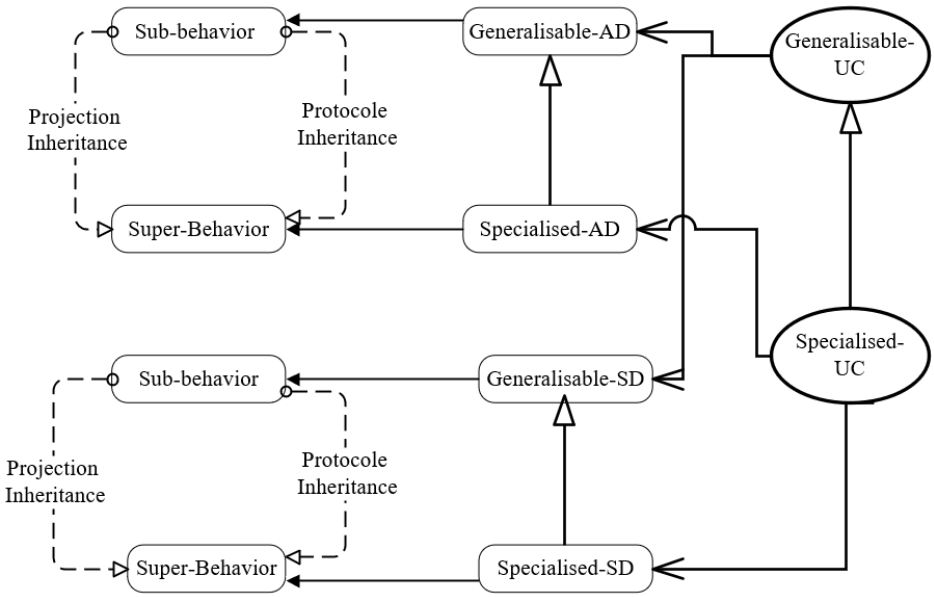

Figure 2. Sub- and Super-behaviour relationship

4. PROPOSED APPROACH

The approach provides flexibility for consistency checking in a UC inheritance framework by affecting AD and SD. It is a combination of definition and translation of consistency rules. For this, we proposed three levels to cover the coherence of the entire system (UC, AD, SD). It will support the following steps: (i) Defining consistency rules between the three diagrams on a vertical level which takes into account the parental specification, (ii) Setting consistency rules on a horizontal level, defining the inherited consistency rules, and (iii) a translation to the SPIN [12] Model Checker for a dynamic consistency checking of the system (see Figure 3).

Layers H1, H2, H3, H4, H5, H6, and H7 contribute to the horizontal level, and layers V1, V2, V3, V4, V5, V6, V7, and V8 provide the vertical level (see Table 4).

We have isolated classes C1 and C2 that oversee the inherited and parental consistency rules, respectively, to put the spotlight on consistency management. Class C2 inherits layers H1, H2, and H3 from the generalizable diagram, while layers H4, H5, and H6 are defined with new rules, especially those that reduce the effectiveness of triggers, preconditions, and postconditions. Class C3 considers layers V1, V2, V3, V4, V5, V6, V7, V8 and V9; class C4 defines the manual transformation rules T1, T2, T3, T4, T5, and T6. It generates the concurrent processes from Büchi automata translation via application to the activity diagram, and generates the assertions via manual transformation and the claim property of the interaction diagrams (see Table 5).

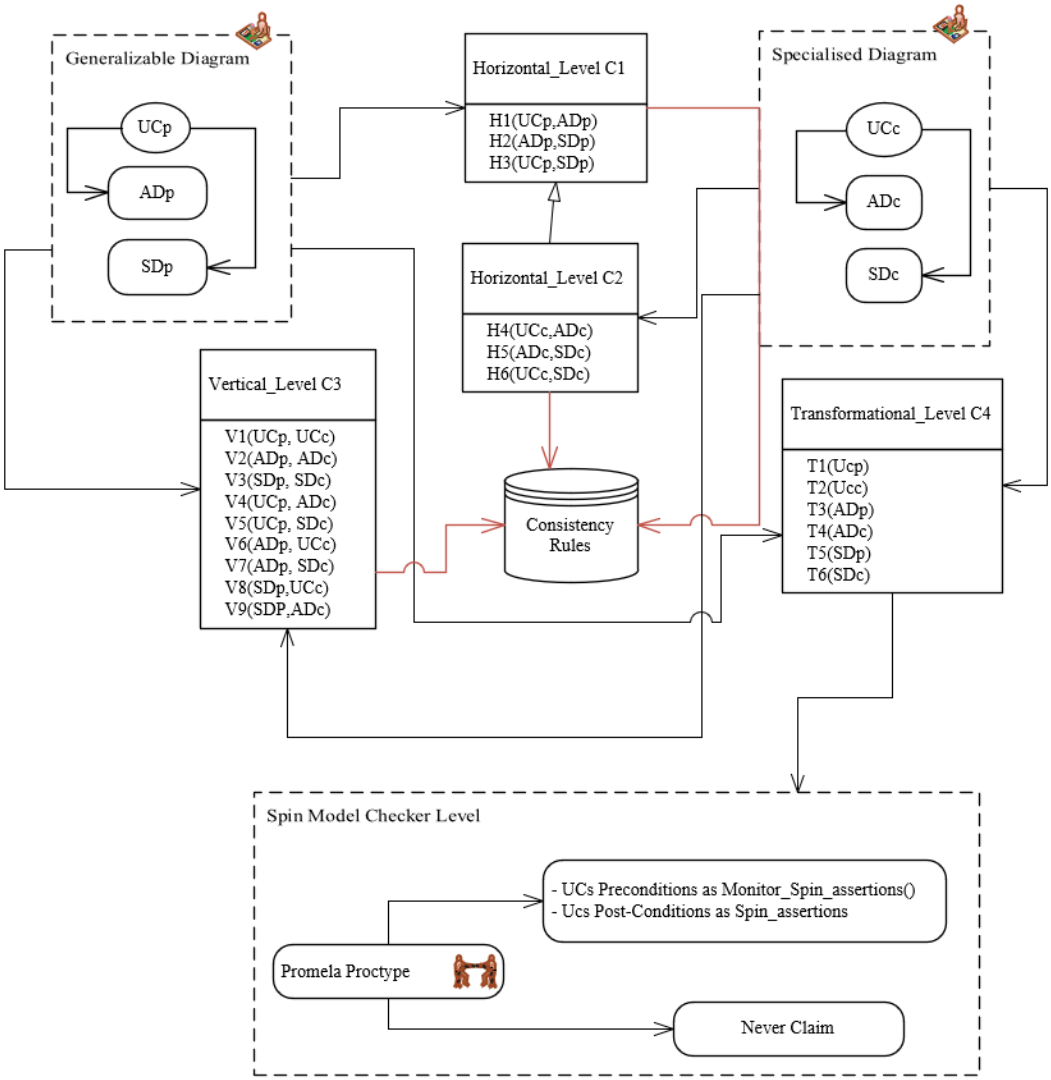

Figure 3. Multi-layer UML consistency checking

Table 5. Layer relation approach

Dia / Diag	Horizontal and Vertical Level					
	UCp	**ADp**	**SDp**	**UCc**	**ADc**	**SDc**
UCp	RWF	H1	H2	V1	V4	V5
ADp	H1	RWF	H3	V6	V2	V7
SDp	H2	H3	RWF	V9	V8	V3
UCc	V1	V6	V9	RWF	H4	H5
ADc	V4	V2	V8	H4	RWF	H6
SDc	V5	V6	V3	H5	H6	RWF
	Transformational Level					
Assert	T1	T3	x	T2	T4	x
Never claims	x	x	T5	x	x	T6

When the corresponding value in Table 4 is RW, this value covers all the consistency rules defined in the UML specification and relates to the same diagram. It has been noted respectively in Table 5 (Ucp, Ucp), Table 4 (Ucc, Ucc), Table 5 (ADp, ADp), Table 5 (SDp, SDp), Table 5 (ADc, ADc), Table 5 (SDc, SDc). The other values define all the intra-diagram consistency rules defined in the literature. For instance, Table 5 (UCp, ADp) = H1 layer represents the consistency rules between the generalizable UCp and ADp diagrams and Table 5 (UCp, ADp) = V1 layer represents the consistency rules between the generalizable UCp and ADp diagrams. The transformation level defines the translation of triggers defined in use cases, activity diagrams, and sequence diagrams from the generalized and specialized levels to Promela. T1 and T2 are associated with UC, T3 and T4 are associated with AD, and T5 and T6 are associated with SD.

It should be noted that Table 5 has been created as a template for future work. Its objective is to collect all the UML consistency rules encountered in the literature and classify them by level and layer, thus enabling the development of a tool that recognizes all these rules and thus ensuring a consistent environment for modellers.

4.1 Transformational level

The behaviour of an AD is controlled by the constraints described in the IoT UC, which may be broken down into three categories: triggers, preconditions, and postconditions. we associate this with the terminology *Correctness Criteria.*

It is usual to represent correctness criteria as a set of Boolean conditions that must be met whenever a process reaches a certain state. The assert(condition) statement will always be run, despite of where it is located within a PROMELA. If the condition is true, the claim is false.

We manually transform each predicate into assertion Promela code. Triggers, postconditions, and preconditions are all considered predicates. The figure's approach defines six transformation rules: T1, T2, T3, T4, T5, and T6.

- T1 is a manual transformation for the trigger predicate of the general UC to a Promela code denoted *assert (UCp.trigger).*

- T2 is a manual transformation for the precondition predicate of the general UC to a Promela code denoted *assert (UCp.pre-condition).*

- T3 is a manual transformation for the trigger predicate of the general use case to a Promela code denoted assert *(UCp.post-condition)*

- T4 contains the new rules that ensure consistency between the child UC the corresponding child AD. *assert (UCc.trigger)*

- T5 contains the new rules that ensure consistency between the child UC and the corresponding child SD. *assert (UCc.precondition)*

- T6 contains the new rules that ensure consistency between the child AD and the corresponding SD. *assert (UCc.postcondition)*.

4.2 Informal properties

A relation that includes the generalizable IoT use case does not include the IoT inherited case.

If an IoT actor B inherits from an IoT actor A, all the IoT UC associated with B cannot inherit from IoT UC associated with A.

When an IoT actor B inherits from an IoT actor A, all IoT UC associated with B cannot include from IoT UC associated with A.

When an IoT actor B inherits from an IoT actor A, all IoT UC associated with B cannot extend from IoT UC associated with A.

When actor an IoT B inherits from an IoT actor A, B cannot be associated with IoT UC associated with A.

The initial activity edge must have the corresponding trigger's IoT UC as guards.

The trigger associated with the initial Activity Edge of Activity diagram match that of the inherited IoT UC and not alter the trigger of the parent IoT UC (to be weakened).

When an activity ends, the postcondition identified in the IoT UC must be true.

An activity diagram Au_i extends an activity diagram Au_j iff all nodes of Au_i are included in Au_j.

The set of preconditions for an action j associated with Au_i must contain the preconditions associated with the IoT UC_i.

The set of postconditions for an action j associated with Au_i must contain the postconditions associated with u_i.

5. MODEL CHECKING

In this section, we discuss utilising SPIN to ensure consistency between UC, AD and SD and the translation of AD into ε-BA [7]. SPIN uses a thorough search of the state space to validate or invalidate (through the generation of counterexamples) the properties of Promela specifications. Another helpful feature is the ability to define the *never claim,* a process for expressing undesirable behaviours.

Model checking will use the layers defined at the previous layers (C1, C2, C3, C4) and the transformation layer, which contains six levels from T1 to T6.

A diagram with UC, AD and SD is consistent if the triggers, preconditions and postconditions of a use case are fulfilled in AD and if the AD's trace collection contains the SD's trace. i.e., iff $trace(Sds) \in trace(Ads)$. It comprises a set of concurrent processes constantly exchanging information with one another and each of these processes represents ε_BA, and the *never claim* to represent the trace of SD. If the system's SPIN report validates the assertions and confirms the never claim then we will know that the SD are consistent with the UC, and AD, elsewhere we know there must be inconsistency. The modeller must inspect the invalid traces to report the errors in requirement models.

5.1 From AD to stutter Büchi automaton (ε-BA)

AD have labels for actions, while ε-BA has labels for transitions. Therefore, during translation, the mapping flips the roles of an AD's nodes and activity edges so that the nodes of the AD represent ε-BA transitions and the activity edges represent ε-BA states. The powerset of the AD's set of activity edges determines the state set of ε-BA. The powerset is required since an AD might exist in multiple nodes at once (after visiting a fork node).

Nodes whose $E_{in}(N_i) = I_i$ represent the initial state of ε-BA and nodes whose $E_{in}(N_i) = F_i$ represent the final states of ε-BA. Labels for nodes in the AD are equal to those for transitions in ε-BA.

As a process can stay infinitely long, a state of the negation of the conjunction of all outgoing events is then added.

The translation conditions associated with the fork, join and merge node are those used in the study of Thramboulidis and Christoulakis [3].

AD have labels for actions, while ε-BA has labels for transitions. Therefore, during translation, the mapping flips the roles of an AD's nodes and activity edges so that the nodes of the AD represent ε-BA transitions and the activity edges represent ε-BA states. The powerset of the AD's set of activity edges determines the state set of ε-BA. The powerset is required since an AD might exist in multiple nodes at once (after visiting a fork node).

Nodes whose $E_{in}(N_i) = I_i$ represent the initial state of ε-BA and nodes whose $E_{in}(N_i) = F_i$ represent the final states of ε-BA. Labels for nodes in the AD are equal to those for transitions in ε-BA.

As a process can stay infinitely long, a state of the negation of the conjunction of all outgoing events is then added.

The translation conditions associated with the fork, join and merge node are those used in Sundaramoorthy [24].

The ε-BA associated to ADi is defined as εBA (ad) = (Q; Σ; δ; q_0; F_0) *where,*

- $Q = \mathbb{P}(T)$,
- $\sum = L$,
- $q_0 = \{(s, t) \in T \mid s = i\}$,
- $F' = \{\{(s, t)\} \subseteq T \mid \{\}\ T / t \in F'\}$, *and*
- $\delta \subseteq Q \times (\Sigma\ U\{\varepsilon\}) \times Q$ is the smallest set satisfying the following conditions:
  - Move other than fork or join:

$\forall X \subseteq T: \forall n_1, n_2, n_3 \in N: ((n_1, n_2) \in X \wedge (n_2, n_3) \in T \wedge n_2 \notin AND) \Rightarrow (X, l(n_2), X \setminus \{(n_1, n_2)\} \cup \{(n_2, n_3)\} \in \delta$

  - Move fork or join *:*

$$\forall X \subseteq T: \forall j \in AND : (\delta^{-}(j) \subseteq X) \Longrightarrow (X, \varepsilon, (X \setminus \delta^{-}_{ad}(j)) \cup \delta^{+}_{ad}(j) \in \delta$$

- $neg_i: Q_i \rightarrow SL_i \cup \{\varepsilon\}$ is the node labeling associates to each state Q_i a self-loop transition (SLi) which contains the negation of the conjunction of all the outgoing states of SL_i.

The trace semantics of an activity diagram *trace_sem_ad* (*AD)* is defined as the language recognized by the *εBA* associated to the AD *ad*, *i.e., trace_sem* (*AD*) =$\mathcal{L}(\varepsilon BA(AD))$.

5.2 Translation Büchi automaton into promela processes

The AD represented by BA are the communication of several objects along several lanes of the AD. The resulting processes can communicate using channels (Chan Keyword) of size 0 to send messages (! Symbol) and receive messages (? Symbol) via their corresponding channels which are then saved in an enumeration set called mtype.

We declare a global variable msgrecord which records each event (message) transferred through a communication channel which we will use as a basis for determining the execution trace of an SD.

The enumeration set mtype records all states and all events for the ε-BA. Each state-related variable begins with the letter S, transitions begin with the letter t, and negations of events begin with the letter NT. if the state is (s, li) the variable Sil1 corresponds to the state(s_il1), if a transition is *log* the associated variable is *Tlog* and the negation transition is *NTlog*.

In the ε-BA, the control flow is set up by a do loop with a local variable *state_automaton* that goes through the list of states $S = \{S_{1..}S_n\}$.

The channel associated with a process that inherits n other processes will be of the form: Chan C_n [0] of {mtype} n times. The process will only be activated if all the associated values are available. For instance, if the process inherits from two others, it has two channels C_1 and C_2.

5.3 Translation from SD to Promela

Trace_sem (SD) is defined as the language recognized by the *ε-BA* associated to the AD, *i.e., trace_sem_sd* (*SD*) = $\mathcal{L}(\varepsilon BA(SD))$ which represents the Promela never claim property.

We can now convert each event into a loop using a guard because we recorded each event with the help of the variable event (event==message). When the guard evaluates to true, this indicates that the event has already taken place; in this scenario, we exit the loop and try to match the subsequent event; nevertheless, in all other cases, the process must wait. If we successfully reach all events, the trace_sem (SD) and trace_sem (AD) will be consistent, as will diagrams, i.e., trace_sem (SD) $\in trace\_sem(ADs)$. Stutter events are ignored and not registered; they are only used to change states.

5.4 System invariants

The assert statement can be used more generally to formalise system invariants, which are *boolean* conditions that, if true in the initial system state, hold true in all reachable system states. In PROMELA, place the system invariant in a separate monitor process.

Proctype control_precondition () {assert(invariant)}.

6. CASE STUDY

The purpose of this system is to provide immediate medical aid to individuals by utilising various forms of technology. We use a portion of the application architecture [20], to which we have integrated IoT services by utilising diabetes and blood pressure sensors. This application allows us to execute the approach concepts; in particular, the communication between the sensors and the smart doctor. It will be used to develop a precise diagnosis of serious diseases caused by the combination of diabetes and high blood pressure, which can even affect pregnant women through the deformation of their babies. The presented automata concern the AD of the diabetes sensor, the hypertension sensor, the composite diagram of the two diseases, and the activities corresponding to the resulting diseases (Hypertensive retinopathy, diabetic nephrology, hypertensive nephrosclerosis, baby's deformity) (see Figure 4).

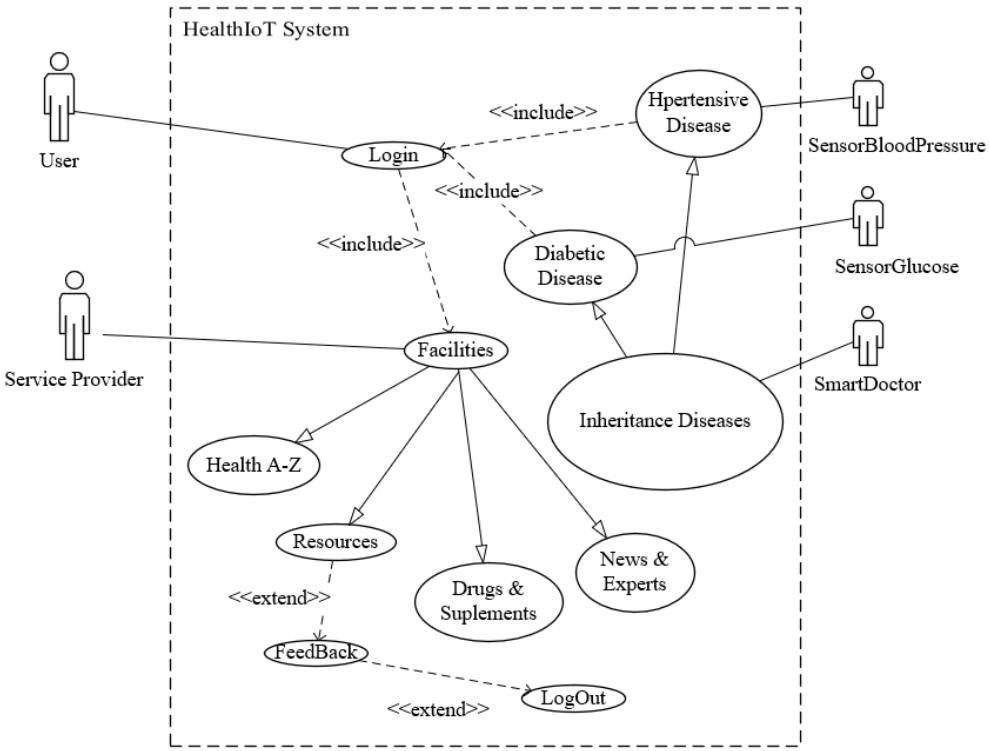

Figure 4. UML UC for HealthIoT system

All the informal properties have been verified.

It will determine the probability of the diseases (Hypertensive retinopathy, diabetic nephrology, hypertensive nephrosclerosis, baby's deformity) being affected.

The diabetes sensor and the blood sensor start after a login from the corresponding patient by executing the action "Patient has glucose test," which will be transmitted to the diabetes doctor by t! phgt and similarly by the blood sensor using t! phpb or the diabetes doctor.

In addition to features *Login, Facilities, Logout, Health A-Z, Resources, Drugs and supplements, New and Experts end Feddback* presented in Sundaramoorthy [24], we will present the following features: The system's primary features are listed below:

Diabetic Disease: This feature allows blood glucose measurements to be taken during patient activities and stored in a data set and predict probability of diabetes and its type. see Figure 5. Figure 6 shows the corresponding Büchi automaton.

Hypertensive Disease: This feature collects patient data, usually by fixed times, and puts them in a data set and predict probability of hypertensive and its treatment. see Figure 7. The Büchi automaton in question is depicted in Figure 8.

The documentation for the blood sugar sensor is provided in Table 6.

Inheritance Diseases: is an intelligent function that inherits hypertensive and Diabetic Diseases for the probability of Hypertensive retinopathy, diabetic nephrology, hypertensive nephrosclerosis, and baby's deformity based on the required patient data.

- Initial Node ={i}
- Final Node = {FN1, FN2, FN3}
- Decision and Merge Nodes XOR= {D1, D2}
- Fork and Join Nodes AND= {F1, J1}
- The transition relation T={(I, F0), (F0, L1), (F0, L2), (L1, PHGT), (PHGT, D1), (D1, FN1), (D1, FN1), (D1, DLHD), (DLHD, F1), (F1, AFT1), (F1, ADPM), (AFT1, J1), (J1, DDT) , (DDT, D2), (D2, DT1), (D2, DT2), (DT1, M1), (DT2, M1), (M1, F3), (F3, HDHR), (F3, HDNHN), (F3, BD), (HDHR, J3), (HDNHN, J3), (BD, J3), (J3, FN3), (L2, PHBPT), (PHBPT, D3), (D3, FN2), (D3, DLHH), (DLHH, F2), (F2, ADT2), (F2, ALSM), (ADT2, J2), (ALSM, J2), (J2, F3), (F3, HDNHN), (F3, BD), (HDHR, J3), (BD, J3), (J3, FN3)}
- Labelling function *l* with l(L1) =Login; l(PHGT)=Patient Has Glucose Test; l(DLHD)= Determine Likelihood of Having diabetes; l(AFT1) = Advice for Test; l(DDT) = Determine diabetes Type; l(DT1) = Display Type 1; l(DT2) = Display Type 2; l(ADT2) = Advice for Test; l(APM)= Advice Proper Meal; L(i)=l(FN1) =l(FN2) =l(D1) =l(D2) =l(F1) =l(F2) =ε.

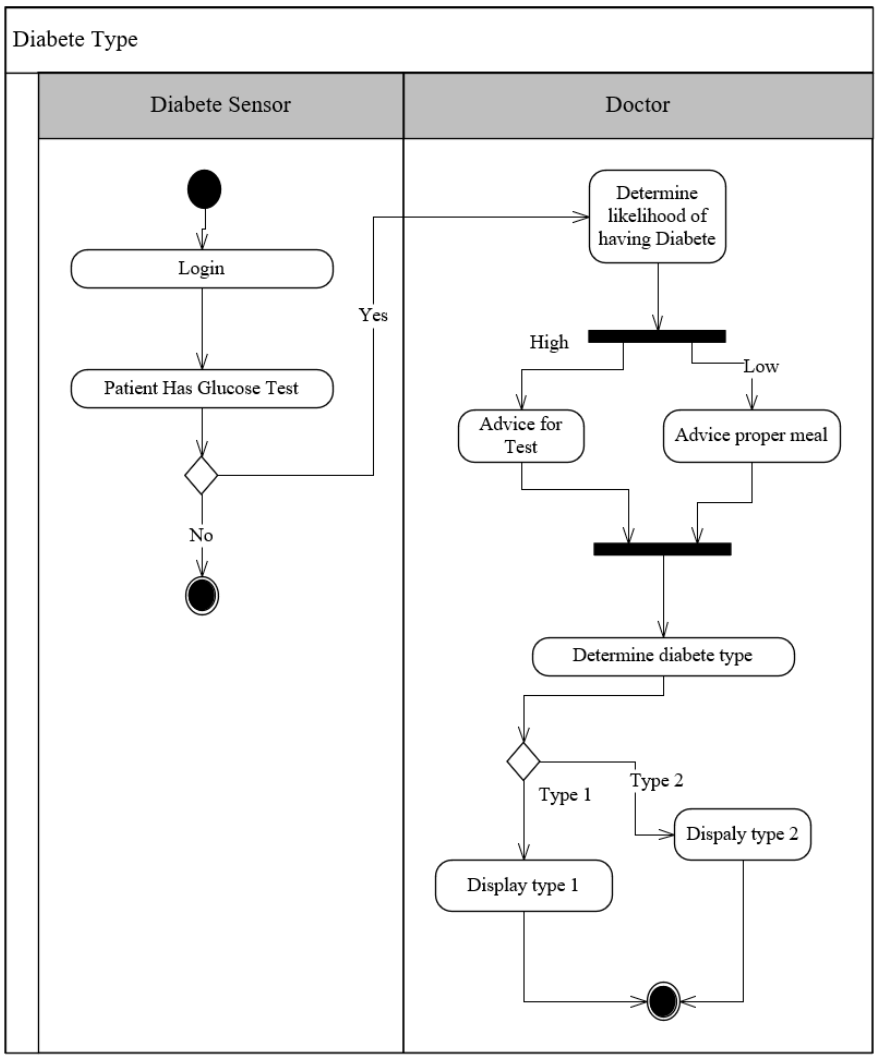

Figure 5. AD for UC Diabetic

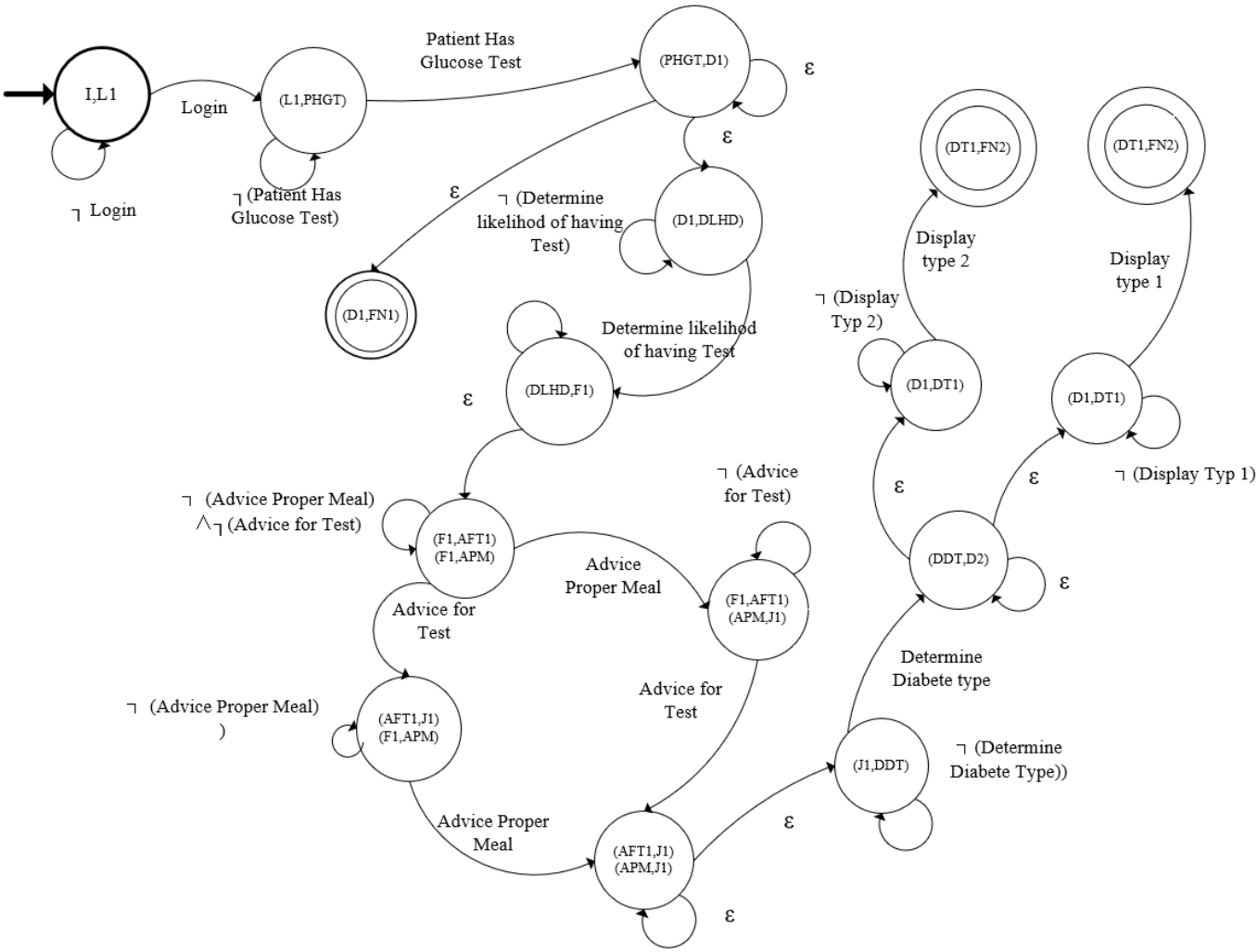

Figure 6. Automaton for diabetic AD

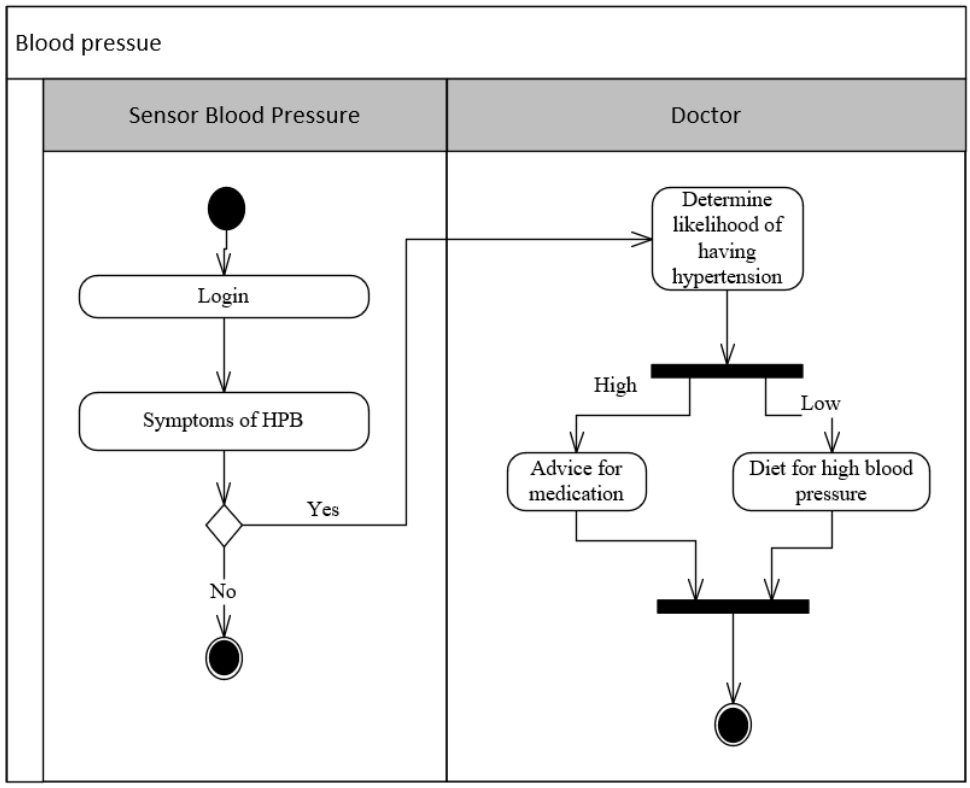

Figure 7. AD for UC Blood Pressure

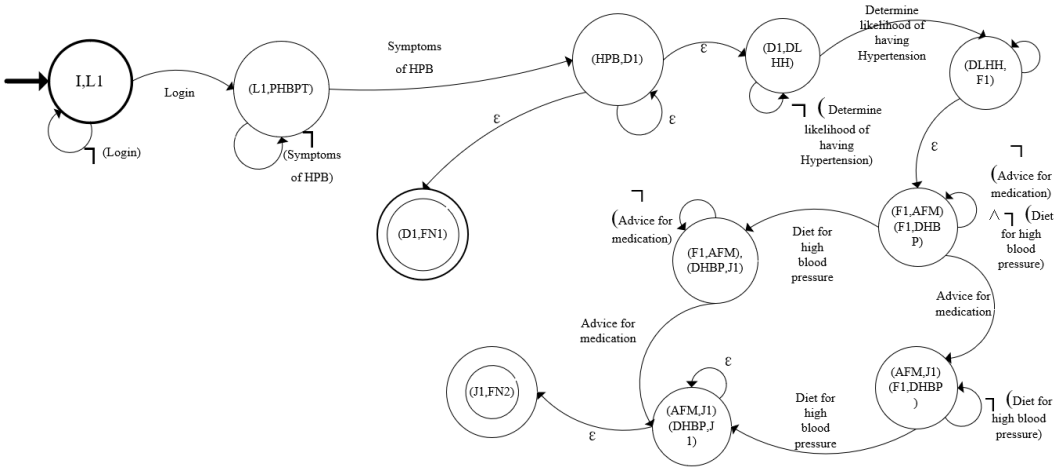

Figure 8. Automaton for AD Blood Pressure

Table 6. Blood sugar sensor

Use Case n°1	**Diabetic Disease**
Goal in Context	The aim is to enable clients of the IoT system to take the corresponding blood sugar value and transmit it to the data set and to give probability of having diabetes.
Scope & Level	Connection Level
Primary/Secondary IoT Actors	Primary: Service Provider
IoT Trigger	After every meal.
Stakeholder &Interest	User
Preconditions	Sensor Available
Postconditions on success	The test sample is sent to the doctor responsible for diabetes.
Postconditions on failure	The system is alerted to the sensor malfunction.
IoT UC Description	1- check mealtimes 2- read the blood sugar 3- transmit the reading.
Alternative description	-The patient can change the sensor responsible for taking the test.
Exceptions	-The information sent is invalid.
IoT UC relationships	-Login

Table 7. Blood pressure

Use Case n°2	Hypertensive Disease
Goal in Context	The aim is to enable patients of the IoT system to take the corresponding blood pressure value and transmit it to the attending doctor for decision-making.
Scope & Level	Connection Level
Primary/Secondary IoT Actors	Primary: Service Provider
IoT Trigger	After every meal.
Stakeholder &Interest	User
Preconditions	Sensor Available
Postconditions on success	The test is sent to the doctor for high blood pressure.
Postconditions on failure	The system is alerted to the sensor malfunction.
IoT use case description	1- check mealtimes 2- the blood pressure level 3- transmit the reading.
Alternative Description	-The patient can change the sensor responsible for taking the test.
Exceptions	-The information sent is invalid.
IoT UC relationships	-Login

Table 8. hypertensive/diabete

Use Case n°3	**Inheritance Disease**
Goal in Context	The aim is to alert patients to the risk of developing a serious illness due to diabetes or high blood pressure.
Scope & Level	Provider (Smart doctor)
Primary/Secondary Actors	Primary: Smart Doctor Secondry: patient
IoT Trigger	Data on blood sugar levels and arterial pressure are available.
Stakeholder &Interest	Patient
Preconditions	Major risk has been reached.
Postcond. On success	Available services.
Postcondit.on failure	Alert the patient that the data is not arriving.
IoT UC Description	1-likelihood of a major risk of illness. 2-Hypertensive retinopathy 3-Diabetic nephrology 4-Hypertensive nephrosclerosis 5-Baby's deformity
Alternative Description	Alert the patient that the data is not arriving.
Exceptions	The required information is invalid
IoT UC relationships	Inherited UC (Sugar & Blood pressure Level).

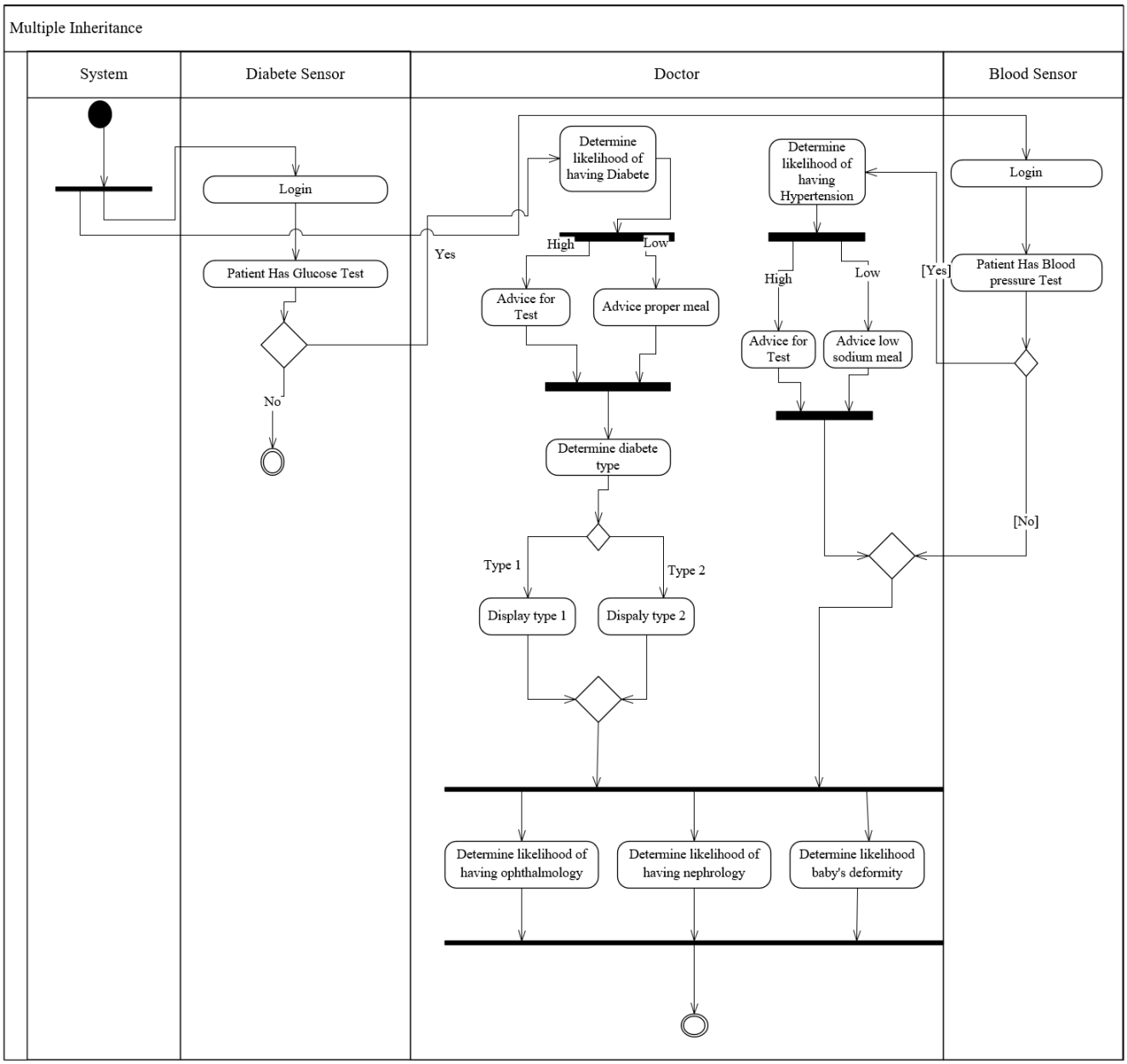

Figure 9. AD for multiple inheritance

Table 7 represents the documentation for the Blood pressure sensor.

-Initial Node ={i}

-Final Node = {FN1, FN2}

-Decision and merge nodes XOR= {D1}

-Fork and join nodes AND= {F1, J1}

-Labelling function l with l(L1) =Login; l(PHBPT)=Symptoms of HPB; l(DLHH)= Determine Likelihood of Having Hypertension; l(AFM)= Advice for Medication; l(DHBP)= Diet for High Blood Pressure; l(i)=l(FN1) =l(FN2) =l(D1) =l(F1) =l(J1) =ε

-The transition relation T= {(I, L1), (L1, PHBPT), (HPB, D1), (D1, FN1), (PHGT, D1), (D1, FN1), (D1, DLHH), (DLHH, F1), (F1, AFM), (F1, DHBP), (F1, ARM), (DHBP, J1), (AFM, J1), (F1, DHBP), (D1, FN1), (J1, FN2)}.

Table 8 shows the documentation for the inheritance Blood sugar sensor and Blood Pressure sensor. The Figure 9 presents the AD pertaining to this UC.

- Initial Node ={i}

- Final Node = {FN1, FN2, FN3}

- Decision and merge nodes XOR= {D1, D2, D3, M1}

- Fork and join nodes AND= {F0, F1, J1, F2, J2, F3, J3}

- The transition relation T={(I, F0), (F0, L1), (F0, L2), (L1, PHGT), (PHGT, D1), (D1, FN1), (D1, FN1), (D1, DLHD), (DLHD, F1), (F1, AFT1), (F1, ADPM), (AFT1, J1), (J1, DDT) , (DDT, D2), (D2, DT1), (D2, DT2), (DT1, M1), (DT2, M1), (M1, F3), (F3, HDHR), (F3, HDNHN), (F3, BD), (HDHR, J3), (HDNHN, J3), (BD, J3), (J3, FN3), (L2, PHBPT), (PHBPT, D3), (D3, FN2), (D3, DLHH), (DLHH, F2), (F2, ADT2), (F2, ALSM), (ADT2, J2), (ALSM, J2), (J2, F3), (F3, HDNHN), (F3, BD), (HDHR, J3), (BD, J3), (J3, FN3)}

- Labelling function l with l(L1)=Login; l(L2)=Login; l(PHGT)=Patient Has Glucose Test; l(DLHD)= Determine Likelihood of Having diabetes; l(AFT1)= Advice for Test; l(ADPM)=Advice Proper Meal; l(DDT) = Determine diabetes Type; l(DT1)= Display Type 1; l(DT2) = Display Type 2; l(HDHR) = Having Diabetic and Hypertensive retinopathy; l(HDNHN)= Having Diabetic Nephrology and Hypertensive Nephrosclerosis; l(BD)= Baby's Deformity; l(PHBPT)= Patient Has Blood Pressure Test; l(DLHH)= Determine

Likelihood of Having Hypertension; l(ADT2)= Advice for Test; l(ALSM)= Advice Low Sodium Meal; l(i)=l(FN1) =l(FN2) =l(FN3) =l(D1) =l(D2) =l(D3) =l(M1) =l(F0) =l(F1) =l(J1) =l(F2) =L(J2) =L(F3) =l(J3) =ε. (see Figure 10)

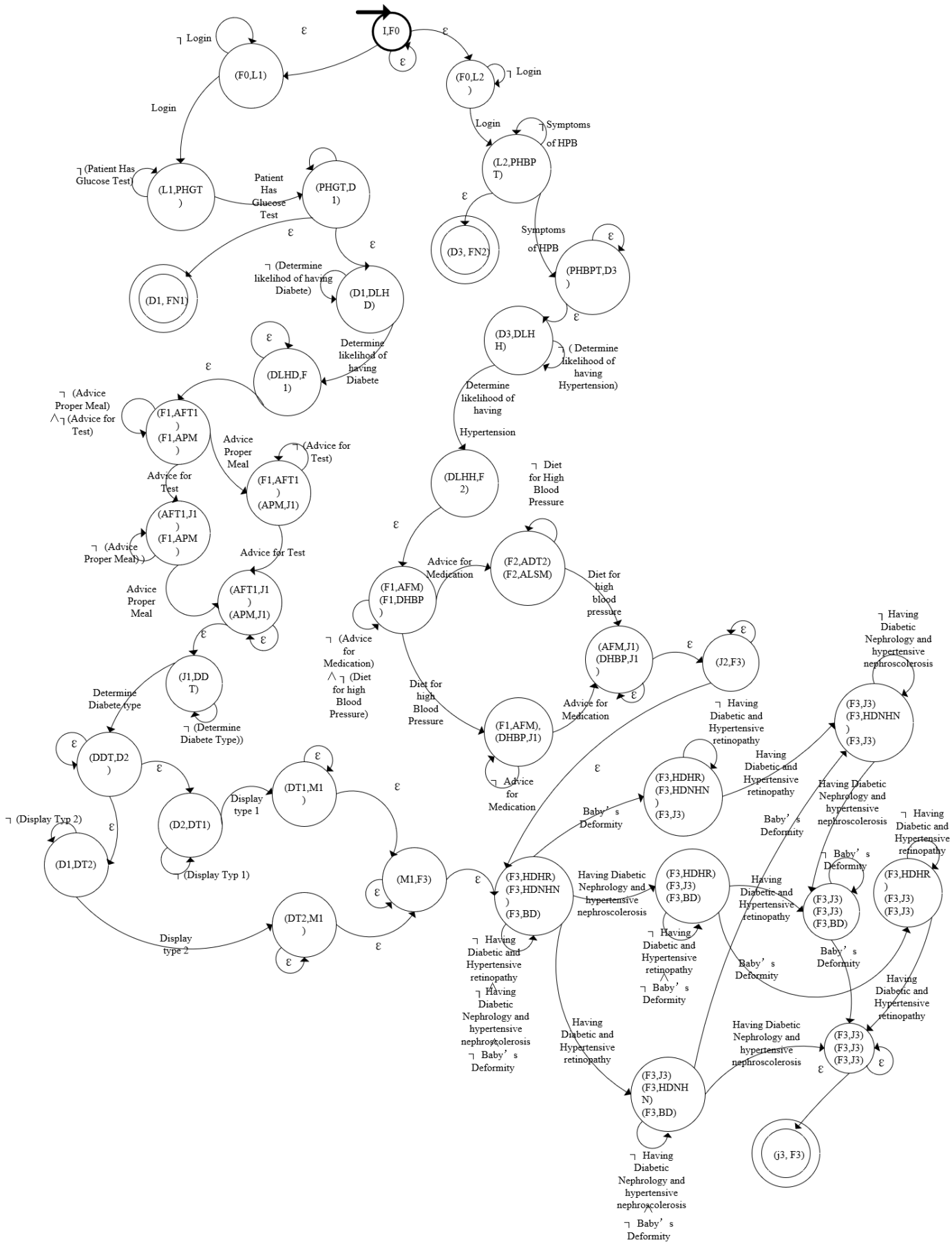

Figure 10. Multiple inheritance automaton

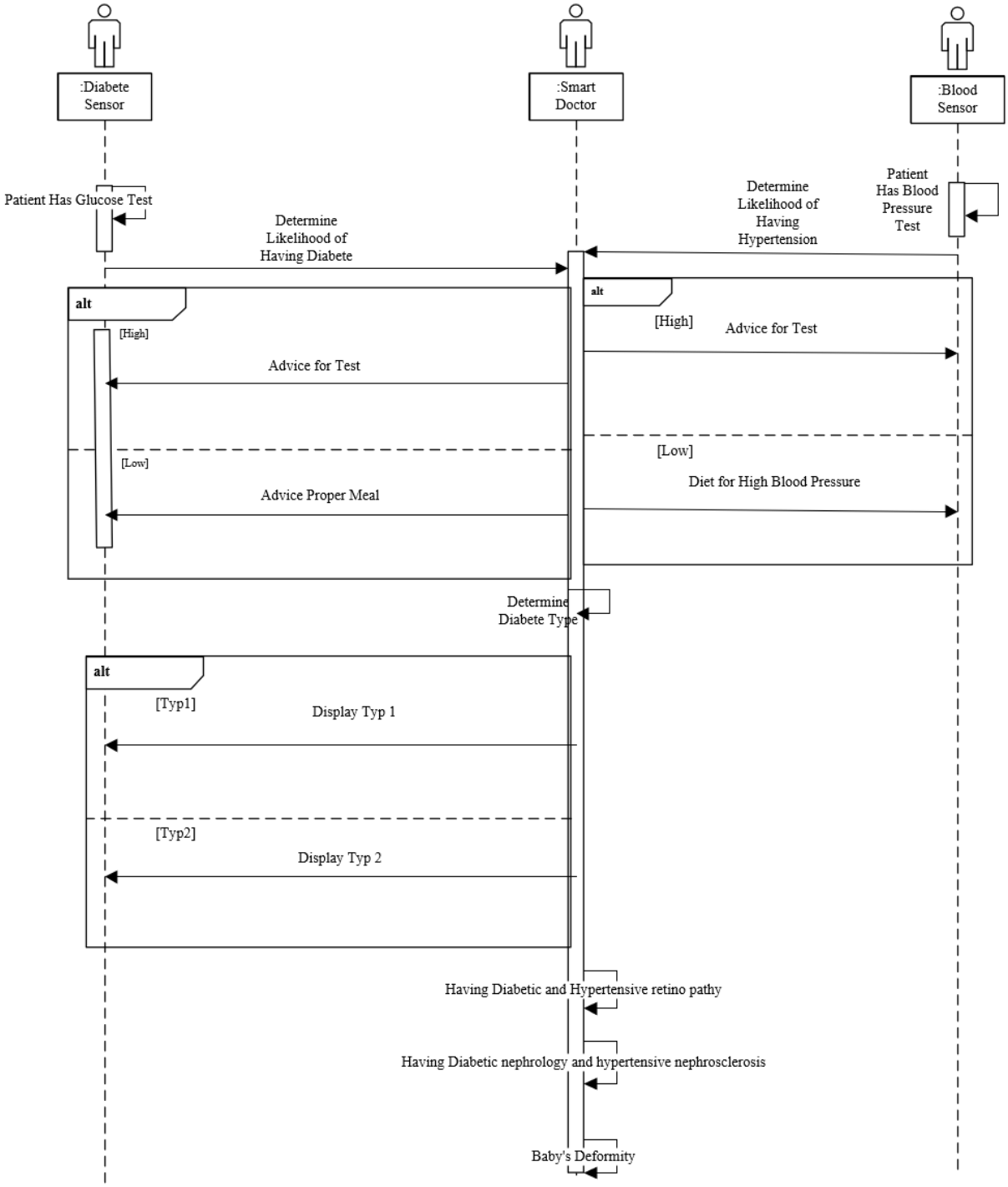

Figure 11. SD for multiple inheritances

We used two timers to simulate the behaviour of the blood sugar and arterial hypertension sensors.

Figure 11 illustrates a valid_trace= (PHBPT, PHGT, DLHH, DLHD, AFM, AFT1, DDT, DT1, HDHR, HDNHN, BD)

The never claim corresponding code:

#define claim_event_message(message) do:: event == message-> break :: else od

never{msg_BA(PHBPT); msg_BA(PHGT); msg_BA(DLHH); msg_BA(DLHD); msg_BA(AFM); msg_BA(AFT1); msg_BA(DDT); msg_BA(DT1); msg_BA(HDHR); msg_BA(HDNHN); msg_BA(BD)}

An invalid trace is a case where the patient has no result for his diabetes, neither type 1 nor type 2, i.e., after DDT, we go on to the HDHR, HDNHN, and BD event.

Invalid_trace= ((PHBPT, PHGT, DLHH, DLHD, AFM, AFT1, DDT, HDHR, HDNHN, BD)

never{msg_BA(PHBPT); msg_BA(PHGT); msg_BA(DLHH); msg_BA(DLHD); msg_BA(AFM); msg_BA(AFT1); msg_BA(DDT); msg_BA(HDHR); msg_BA(HDNHN); msg_BA(BD)}

7. CONCLUSIONS

This work was initiated by articles that present statistics on UML consistency rules. Because of this effort, we have concluded that, despite the significance of inheritance in MDE (Model Driven Engineering), only a few works treat this issue

in MDE. After investigating several works in-depth, we concluded that only some rules were presented to link inherited diagrams.

Although this concept is essential in specification, design and object-oriented programming, we found a need for more work on the inheritance of diagrams. We set out to develop an approach to this topic, from formalisation to model checking.

Our approach uses the template provided by the UC documentation and adapted to the IoT system as a starting point. It uses the relationships (Include extension and generalisation) between UC, the generalisation relationship between actors, and the association between them and UC, especially the triggers, preconditions and postconditions of UC.

The proposed approach uses the horizontal levels H1, H2, H3, H4, H5, H6 and the vertical levels V1, V2, V3, V4, V5, V6, V7, V8, V9. Inheritance is supported in the vertical levels. To use the SPIN, we used the *Split Automata* algorithm, where the sequence diagrams' interactions are considered *Never Claim* property.

In addition, completing this work has made it possible to design additional consistency rules according to the horizontal level and others according to the vertical level.

We have assessed the proposed method based on the IoT case study by determining how helpful the examined UML models, consistency rules, and model types are in determining whether or not UML models are consistent.

In addition to this, we investigated how SPIN makes it possible to evaluate inconsistencies that may arise during the various stages of software development.

In the Appendix, you'll see that we've transformed both the activity AD and the SD into Promela codes. We have confirmed the SD's consistency using SPIN. Our approach is successful and efficient, as the verification process only takes a few seconds (see Appendix).

Our inherited consistency rules will be encoded in OCL and compared to existing OCL consistency rules. Using ATL (Atlas Transformation Language) to generate Promela assertions Code automatically. We will also gather and study UML models not developed with Eclipse Metamodeling Approach.

In future work, we also envisage taking into account when defining the automata of the alternative exceptions, postconditions on failure and exceptions of the UC documentation.

ACKNOWLEDGMENT

We want to thank our laboratory ICOSI of Khenchela for its total availability in terms of documentation (Books, Articles) and the advice of the teachers of the mathematics and computer science department of the university Abbes Laghrour, Khenchela, Algeria.

We want to thank the team from the English literature department at the Abbes Laghrour University in Khenchela for their assistance throughout the production of this article.

The staff of the Misc of Constantine 2, Algeria, is also thanked for their kind help.

REFERENCES

[1] Rumbaugh, J., Ivar, J., Grady, B. (2010). The Unified Modeling Language Reference Manual. Addison-Wesley Professional; 2nd edition.

[2] Ciccozzi, F., Spalazzese, R. (2016). Mde4iot: Supporting the internet of things with model-driven engineering. In International Symposium on Intelligent and Distributed Computing, Paris, France, pp. 67-76. https://doi.org/10.1007/978-3-319-48829-5 7

[3] Thramboulidis, K., Christoulakis, F. (2016). UML4IoT-A UML-based approach to exploit IoT in cyber-physical manufacturing systems. Computers in Industry, 82: 259-272. https://doi.org/10.1016/j.compind.2016.05.010

[4] Bashir, R.S., Lee, S.P., Khan, S.U.R., Chang, V., Farid, S. (2016). UML models consistency management: Guidelines for software quality manager. International Journal of Information Management, 36(6): 883-899. https://doi.org/10.1016/j.ijinfomgt.2016.05.024

[5] Shinkawa, Y. (2006). Inter-model consistency in UML based on CPN formalism. 2006 13th Asia Pacific Software Engineering Conference (APSEC'06), Bangalore, India. https://doi.org/10.1109/apsec.2006.41

[6] van der Aalst, W.M. (2002). Inheritance of dynamic behaviour in UML. MOCA, 2: 105-120.

[7] Kautz, O., Rumpe, B. (2018). Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In MoDELS (Workshops), pp. 574-583.

[8] Almendros-Jiménez, J.M., Iribarne, L. (2007). Describing use-case relationships with sequence diagrams. The Computer Journal, 50(1): 116-128. https://doi.org/10.1093/comjnl/bxl053

[9] Ibrahim, N., Ibrahim, R., Saringat, M.Z., Mansor, D., Herawan, T. (2011). Consistency rules between UML use case and activity diagrams using logical approach. International Journal of Software Engineering and its Applications, 5(3): 119-134.

[10] Chanda, J., Kanjilal, A., Sengupta, S., Bhattacharya, S. (2009). Traceability of requirements and consistency verification of UML use case, activity and Class diagram: A Formal approach. In 2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS), New Delhi, India, pp. 1-4. https://doi.org/10.1109/icm2cs.2009.5397941

[11] Torre, D., Genero, M., Labiche, Y., Elaasar, M. (2023). How consistency is handled in model-driven software engineering and UML: an expert opinion survey. Software Quality Journal, 31(1): 1-54. https://doi.org/10.1007/s11219-022-09585-2

[12] Holzmann, G.J. (2004). The SPIN model checker: Primer and reference manual. Reading: Addison-Wesley, 1003.

[13] Torre, D., Labiche, Y., Genero, M., Elaasar, M. (2018). A systematic identification of consistency rules for UML diagrams. Journal of Systems and Software, 144: 121-142. https://doi.org/10.1016/j.jss.2018.06.029

[14] Lima, L., Tavares, A., Nogueira, S.C. (2020). A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP. Science of Computer Programming, 197: 102497. https://doi.org/10.1016/j.scico.2020.102497

[15] Chen, X., Liu, Q., Mallet, F., Li, Q., Cai, S., Jin, Z. (2022). Formally verifying consistency of sequence diagrams for safety critical systems. Science of Computer Programming, 216: 102777. https://doi.org/10.1016/j.scico.2022.102777.

[16] Doostali, S., Babamir, S.M., Javani, M. (2023). Using a process algebra interface for verification and validation

of UML statecharts. Computer Standards & Interfaces, 86: 103739. https://doi.org/10.1016/j.csi.2023.103739
[17] Kalibatiene, D., Vasilecas, O., Dubauskaite, R. (2013). Rule based approach for ensuring consistency in different UML models. In Information Systems: Development, Learning, Security: 6th SIGSAND/PLAIS EuroSymposium 2013, Gdańsk, Poland, pp. 1-16. https://doi.org/10.1007/978-3-642-40855-7_1
[18] Kautz, O., Rumpe, B., Wachtmeister, L. (2022). Semantic differencing of use case diagrams. Journal of Object Technology, 21(3): 1-14. https://doi.org/10.5381/jot.2022.21.3.a5
[19] Sapna, P.G., Mohanty, H. (2007). Ensuring consistency in relational repository of UML models. In 10th International Conference on Information Technology (ICIT 2007), Rourkela, India, pp. 217-222. https://doi.org/10.1109/icit.2007.43
[20] Lee, R.Y. (2019). Object-Oriented Software Engineering with UML: A Hands-on Approach (Computer Science, Technology, and Applications). Nova Science Pub Inc.
[21] Stephens, M., Rosenberg, D. (2007). Use Case Driven Object Modeling with UML-Theory and Practice. Berkeley: Apress.
[22] Amor, S.O.B., Ali, M., Gargouri, F. (2011). Verification of the consistency between use case and activity diagrams: A step towards validation of user requirements. In 13th International Conference on Enterprise Information Systems, ICEIS 2011, Beijing, China, pp. 396-399. https://doi.org/10.5220/0003505503960399
[23] Stumptner, M., Schrefl, M. (2003). Behavior consistent inheritance in UML. Conceptual Modeling-ER 2000, Salt Lake City, Utah, USA, pp. 527–542. https://doi.org/10.1007/3-540-45393-8_38
[24] Sundaramoorthy, S. (2022). UML Diagramming: A Case Study Approach. Auerbach Publications. https://doi.org/10.1201/9781003287124

APPENDIX

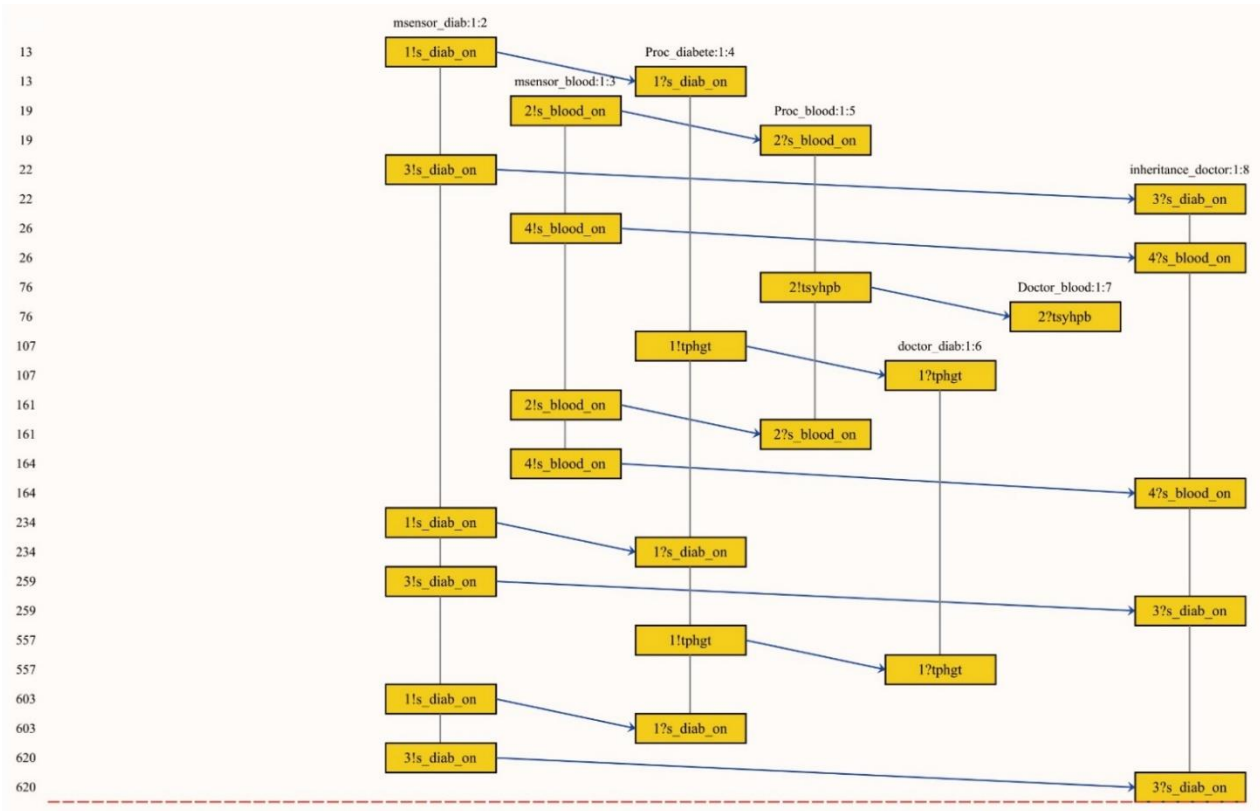

Figure 12. Message Sequence Chart from SPIN

Case Study SPIN Simulation result for 10000 steps. (Figure 12).

depth-limit (-u10000 steps) reached
#processes: 9
10000:proc 8 (inheritance_doctor:1) iot_final_version.pml:251 (state 3)
10000: proc 7 (Doctor_blood:1) iot_final_version.pml:201 (state 77)
10000: proc 6 (doctor_diab:1) iot_final_version.pml:135 (state 110)
10000: proc 5 (Proc_blood:1) iot_final_version.pml:118 (state 23)
10000: proc 4 (Proc_diabete:1) iot_final_version.pml:90 (state 30)
10000: proc 3 (msensor_blood:1) iot_final_version.pml:59 (state 3)
10000: proc 2 (msensor_diab:1) iot_final_version.pml:51 (state 3)
10000: proc 1 (activate_sensor_b:1) iot_final_version.pml:41 (state 5)
10000: proc 0 (activate_sensor:1) iot_final_version.pml:37 (state 6)
9 processes created for 10000 steps.