IAFinder: Identifying Potential Implicit Assumptions to Facilitate Validation in Medical Cyber-Physical System

Zhicheng Fu, Zhao Wang, Chunhui Guo,

Zhenyu Zhang, Shangping Ren

Department of Computer Science

Illinois Institute of Technology

{zfu11,zwang185,cguo13,zzhang111}@hawk.iit.edu,
ren@iit.edu

Lui Sha

Department of Computer Science

University of Illinois at Urbana-Champaign

lrs@illinois.edu

ABSTRACT

According to the U.S. Food and Drug Administration (FDA) medical device recall database, medical device recalls are at an all-time high. One of the major causes of the recalls is due to implicit assumptions of which either the medical device operating environment does not match, or the device operators are not aware of. In this paper, we present *IAFinder* (Implicit Assumption Finder), a tool that uses data mining techniques to automatically extract invariants from design models implemented with statecharts. By identifying invariants that are not explicitly specified in the design models, we are able to find implicit assumptions and better facilitate domain experts to validate them and make the validated implicit assumptions explicit. We use a cardiac arrest statechart model as a case study to illustrate the usage of *IAFinder* in identifying implicit assumptions.

ACM Reference Format:

Zhicheng Fu, Zhao Wang, Chunhui Guo, Zhenyu Zhang, Shangping Ren and Lui Sha. 2018. *IAFinder: Identifying Potential Implicit Assumptions to Facilitate Validation in Medical Cyber-Physical System*. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3195970.3196062>

1 INTRODUCTION

As technology advances, Medical Cyber-Physical Systems (M-CPS) are playing increasingly more important roles in patient care. Unfortunately, according to the U.S. Food and Drug Administration (FDA) medical device recall database, medical device recalls are at an all-time high [5]. One of the major causes of the recalls is due to implicit or unspecified assumptions made about physical or cyber components of the systems. An implicit assumption is an assumption that includes the underlying agreements made in the development of system life cycle that are not explicitly documented. These implicit assumptions are either incorrect or do not match

the environment in which these devices operate. We use the following FDA medical device recall example to illustrate how implicit assumptions have caused an M-CPS failure.

FDA MEDICAL DEVICE RECALL 1. **HAMILTON-T1 ventilators with software versions 1.1.2 and lower. 01/23/2013 [6].** "The FDA has identified this as a Class I recall—the most serious type of recall issued by the FDA. According to the manufacturer, there is unexpected high internal oxygen consumption of HAMILTON-T1 ventilators during ventilation of small pediatric patients. The current labeling does not include sufficient information about the internal oxygen consumption of the ventilators. Depending on the ventilator setting and the patient's lung impedance, the internal gas consumption of the HAMILTON-T1 may be higher than expected. The internal oxygen consumption has to be taken into account in addition to the oxygen requirements for the patient's ventilation, especially when oxygen is limited. If the available oxygen supply during transport is depleted, the life of the patient may be endangered."

In this recall, the internal oxygen consumption is calculated by $(\text{MinVol} + \text{FlowTrigger}) \times C \times (F_i\text{O}_2 - 20.9)/79.1$, where constant $C = 1.5$ is initially set for adult patients – an implicit assumption that the formula of internal oxygen consumption is used for adult patients. For pediatric patients, the value of C should be 4.0, which is different from the value used for adult patients and causes unexpected high internal oxygen consumption. This recall and many other examples in FDA recall database [7] show that implicit assumptions about constant variables can be dangerous and lead to catastrophes. Therefore, being able to identify such assumptions at an early stage of M-CPS development process is critical to ensure the safety of M-CPS.

Unfortunately, implicit assumptions often exist in domain experts' minds, such as $C = 1.5$ in the ventilator recall example. There is a type of information that is critical to the safety of patients, but domain experts are so familiar with that they often do not realize they need to explicitly specify the information. This subconscious habit makes identifying the implicit assumptions dauntingly difficult. Through the analysis of many FDA recall cases [8], we notice that implicit assumptions are often linked with some invariants, such as constant values and execution statement patterns. To finding the implicit assumptions behind constant variables and execution statement patterns at the early stage of system development life cycle, we propose an approach to extricate potential implicit assumptions from design models. In particular, we present an implicit assumptions finder called *IAFinder*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5700-5/18/06...\$15.00

<https://doi.org/10.1145/3195970.3196062>

that uses data mining techniques to automatically extract underlying invariants from design models implemented by Yakindu statecharts. The techniques used by *IAFinder* do not require any prior knowledge about annotations, templates or weight assignments from users, and can automatically extract invariants in the forms of constant variables and execution statement groups in design models.

The rest of the paper is organized as follows: Section 2 discusses related work. In Section 3, we use an example to illustrate the forms of implicit assumptions commonly occur in medical treatment statechart models. Section 4 presents the design of *IAFinder* for automatically extracting invariants from design models implemented by Yakindu statechart. Section 5 uses a cardiac arrest assist system as a case study to evaluate and demonstrate the usage of *IAFinder*. We draw conclusions and future work in Section 6.

2 RELATED WORK

Many tools and frameworks have been proposed to facilitate developing safe M-CPS, such as Petri Nets [20], SimuLink [19], SCXML [1], UPPAAL [2] and Yakindu statechart [12]. With the support of these tools, model-driven approaches, real-time simulation, and formal verification are effectively applied in developing safe and correct M-CPS. However, assumptions about a medical cyber-physical system's expected environment are often informally documented, or even left implicit and unspecified in system design, which leads to many catastrophes [7].

Implicit assumptions is one of the important factors that are determined to be the cause of failures in safety-critical cyber-physical systems [22]. This is experienced in several industry and academic projects. The Ariane 5 [11] and Child-seat Airbag Incident [15] are the results of implicit assumptions made in system design. The problem is magnified further in M-CPS development such as many recent medical device recalls documented in FDA recall database [7]. Much work has been done for the explication of assumptions in system design.

For instance, Lehman and Ramil have proposed several guidelines for managing assumptions [14]. The authors believed that it is necessary to train all stakeholders to identify and record assumptions (including explicit and implicit assumptions) at all stages of the development based on a standard form or structure. In addition, Lewis et al. [16] proposed a prototype Assumptions Management System in software development, which can extract assumptions from source code and record them into a repository for management. To capture mismatched assumptions between software components, an assumption management framework has been introduced by Tirumala with the aim of designing a set of well-defined vocabularies to encode architectural assumptions of a system [3].

In this paper, we present a tool to identify implicit assumptions at the design level and facilitate domain experts to validate them. The techniques used by the *IAFinder* presented in this paper is inspired by Li and Zhou's work that which uses data mining techniques to efficiently extract implicit programming rules from large software code [17]. The major challenge for model level implicit assumptions extraction and

identification is that the assertions and comments are rarely used while at the code level, programmers often use assertions and comments in their code to make the assumptions about invariants explicit [23]. Our strategy to overcome the challenge is through identification of invariants, such as constant variables and execution statement groups, to find potential implicit assumptions.

3 POTENTIAL IMPLICIT ASSUMPTIONS IN STATECHART MODELS

Noting that Yakindu statechart has a high remembrance to disease and treatment models, can be easily understood by medical professionals and is executable, many efforts have been done in using Yakindu statechart to model medical systems [10, 13, 21]. However, how to identify implicit assumptions unintentionally hidden in Yakindu models and help domain experts to validate them is yet to be addressed.

Since the medical device manufacturer would not like to provide the internal design model of the recall example 1 to the public, we use an existing Yakindu statechart for blood gas examination [21] depicted in Fig. 1 as an example to show some implicit assumptions in the model. Blood gases are measurements of how much oxygen and carbon dioxide are in patients bloodstream, which is often used for detecting metabolic acidosis, respiratory acidosis, and hypercapnia [21].

Figure 1: Blood Gas Examine Yakindu Statechart Model [21]

In this statechart model, there are many variables that are initialized with constant values. For example, the variable *BGI.paCO2_High_Threshold* is used as a constant and represents the threshold of the pressure of carbon dioxide (PaCO2) dissolved in the blood. The occurrences of *BGI.paCO2_High_Threshold* are highlighted with red boxes in Fig. 1. The initial value of *BGI.paCO2_High_Threshold* is set 45 because PaCO2 > 45mmHg is the diagnostic indication of metabolic alkaloids for *adult* patients. Here, an implicit assumption is made that the model is for adult patient. If it were used to examine pediatric patients, it would output inaccurate test results and may lead doctors to make incorrect decisions.

In this model, we also notice that there are repeated action group. For example, the action *raise BGI.PatientHasMetabolicAcidosis* is executed together with the action *metabolic_acidosis_hold_timer=0* in a few places as highlighted in blue boxes

in Fig. 1. This repeated action group points to another potential implicit assumption that every time when the diagnosis of metabolic acidosis is confirmed, the holding timer should be reset to zero.

As the example shows, implicit assumptions in the forms of constant variables and action groups often occur at mode level. Identifying these potential implicit assumptions can provide significant help to domain experts in validating the correctness of these assumptions and improve the safety of M-CPS systems.

4 THE ARCHITECTURE AND IMPLEMENTATION OF THE IAFINDER

The design foundation of the *IAFinder* is based on our extensive study of the FDA recall database [8], from which we find that many implicit assumptions are associated with invariants, such as constants, and action groups. Hence, the goal of the *IAFinder* is to identify the constants and action groups by looking for elements that are used as constants, and elements that are frequently used together as possible action groups. In order to achieve the goal, we partition the task into three steps, i.e., 1) parsing statechart model, 2) extracting constant variables, and 3) extracting possible action groups. The architecture of *IAFinder* is depicted in Fig. 2.

Figure 2: The Architecture of *IAFinder*

The following subsections give the details about the implementation of each step.

4.1 Parsing Yakindu Statechart Model

Yakindu statechartmodels are stored as XML files. Fig 3 depicts an example Yakindu statechart model and its XML file.

A Yakindu statechart model consists of the following basic elements: 1) *state*, 2) *variable*, 3) *transition*, 4) *guard*, 5) *event*, and 6) *action*. Actions can occur in both transitions and states as entry actions or exit actions. There are three types of actions in the Yakindu statechart i.e., *assignment* action and *raise* event action, and a special *conditional* action which is used to represent *guard*. To simply parse the XML file of a Yakindu statechart into elements represented by strings and access the conveniently, we define the following structures to represent these five basic elements as below:

- A state s is represented as $\langle id, name, T_{in}, T_{out}, A_s \rangle$, where T_{in} and T_{out} are a set of transition *ids* from and out of a state, respectively, and A_s is a set of actions which are to be executed when the state becomes active.

Figure 3: A Simple Yakindu Statechart Model and Its XML File

- A variable v is represent as $\langle name, type, value \rangle$
- A transition t is represented as $\langle id, s_f, s_e, A_t \rangle$, where s_f and s_e are the *ids* of from and to states of the transition, respectively, and A_t is a set of actions which are to be executed when the transition is active.
- As an event in a Yakindu statechart is implemented as a boolean variable, we use variable structure, i.e., $\langle name, Boolean, value \rangle$ to represent events.
- An action a is represented as $\langle comb, V_c, V_u \rangle$, where *comb* can be either a variable *assignment* or *raise* or *conditional*, V_c is a set of variables whose value will be changed after execution of this action, and V_u is a set of variables that are involved in the action but whose values are not changed.
- A guard is represented as $\langle "conditional", \emptyset, V_u \rangle$.
- A Yaindu statechart model is represented as $Y = \langle name, V, T, S \rangle$, where V, T, S are sets that contain all the variables, the transitions, and the states in the statechart, respectively.

The purpose of the parsing procedure is to instantiate all the structures from a Yakindu statechart model. Since a Yakindu statechart is constructed by using specified XML schemes, we use a DOM parser [18] to parse Yakindu statechart XML document.

However, it is not enough to access only an independent element and extract its information. We need also to record relationship information among the elements, such as which state an action is taken place, and what variables are involved in an action or changed by an action. Such information is critical in identifying constant variables and action groups. Unfortunately, the original DOM parser does not provide the functionality and needs to be extended. Algorithm 1 depicts the procedure for constructing the defined structures from the DOM parser.

Once all the structural elements of a Yakindu statechart statechart model are instantiated, the relationships among variable, action, transition and states can be extracted for analysis.

4.2 Mining Constant Variables

In a Yakindu statechart, a constant variable can be declared with keyword *const*. However, developers usually declare many local variables but use them as constants. Therefore, we

Algorithm 1 YAKINDU STATECHART STRUCTURAL ELEMENTS INSTANTIATIONS

Input: A XML Source File X of a Yakindu Statechart
Output: An Instantiated Yakindu Statechart $Y = \langle name, V, T, S \rangle$

- 1: Parse X with DOM parser to be a DOM structure D
- 2: Declare $Y = \langle name = "", V = \emptyset, T = \emptyset, S = \emptyset \rangle$
- 3: For the root node n_0 of D , set $Y.name = n_0.name$ and partition $n_0.specification$ with Yakindu statechart keywords to instantiate V
- 4: **for** each *vertice* node $n_i \in D$ **do**
- 5: Declare a state $s_i = \langle id = "", name = "", T_{in} = \emptyset, T_{out} = \emptyset, A_s = \emptyset \rangle$.
- 6: Set $s_i.id = n_i.id$, $s_i.name = n_i.name$. Partition $n_i.specification$ to instantiate A_s . Partition $n_i.incomingTransitions$ to instantiate T_{in} . $S = \{s_i\} \cup S$.
- 7: **end for**
- 8: **for** each *outgoingTransition* node $n_j \in D$ **do**
- 9: Declare a transition $t_j = \langle id = "", s_f = "", s_t = "", A_t = \emptyset \rangle$.
- 10: Set $t_j.id = n_j.id$ and $t_j.s_t = n_j.target$. Partition $n_j.specification$ to instantiate A_t .
- 11: Get the parent node n_j^p of n_j , set $t_j.s_f = n_j^p.id$ and $T = \{t_j\} \cup T$.
- 12: Find the state $s \in S \wedge s.id = n_j^p.id$. $s.T_{out} = \{t_j.id\} \cup s.T_{out}$.
- 13: **end for**
- 14: **return** Y

not only need to extract the variables already declared with keyword *const*, but also have to examine the statechart model to extract the variables that are not declared as constants but used as constants. We define two rules to determine whether a variable is a constant variable:

- R.1: A variable is declared with the keyword *const*.
- R.2: A variable is not declared with the keyword *const*, but is used at some actions and not changed by any actions.

To record constant variables and traces where the variables are used, we define a struct $C = \{c | c = \langle v, S_u, T_u \rangle\}$, where v is a constant variable, S_u is a set of states in which v is used, T_u is a set of transitions in which v is used. Given an instantiated Yakindu statechart $Y = \langle name, V, T, S \rangle$, we declare $C = \emptyset$. For each variable $v_i \in V$, we traverse all actions in T and S , if v_i satisfies R.1 and R.2, we declare $c_i = \langle v_i, S_u^i = \emptyset, T_u^i = \emptyset \rangle$ and put the states and transitions in which v_i is used to S_u^i and T_u^i , respectively. At the end we set $C = C \cup \{c_i\}$. Algorithm 2 depicts the procedure extracting constant variables from an instantiated Yakindu statechart model. Through this procedure, the *IAFinder* can extract all constant variables and output the traces where a constant variable is used. Fig. 4 shows the traces where the variable *BGI.paCO2_High_Threshold* is used in the Blood Gas Examination.

Algorithm 2 EXTRACTING CONSTANT VARIABLES

Input: An Instantiated Yakindu Statechart $Y = \langle name, V, T, S \rangle$
Output: A Set $C = \{c | c = \langle v, S_u, T_u \rangle\}$, where v is a constant variable, S_u is a set of states in which v is used, T_u is a set of transitions in which v is used.

- 1: Declare $C = \emptyset$
- 2: **for** each variable $v_i \in V$ **do**
- 3: Declare a boolean variable *isConst* = *true* and $c_i = \langle v_i, S_u^i = \emptyset, T_u^i = \emptyset \rangle$.
- 4: **for** each state $s_i \in S$ **do**
- 5: For each action in s_i , if v_i is used, $S_u^i = S_u^i \cup \{s_i\}$; if v_i is changed, set *isConst* = *false*.
- 6: **end for**
- 7: **for** each transition $t_i \in T$ **do**
- 8: For each action in t_i , if v_i is used, $T_u^i = T_u^i \cup \{t_i\}$; if v_i is changed, set *isConst* = *false*.
- 9: **end for**
- 10: **if** *isConst* **then**
- 11: $C = C \cup \{c_i\}$
- 12: **end if**
- 13: **end for**
- 14: **return** C

Figure 4: Traces where *BGI.paCO2_High_Threshold* Is Used

4.3 Mining Action Groups

In a Yakindu statechart model, multiple actions may be executed together in a transition or a state. By identifying which actions are executed together frequently in the Yakindu statechart model, we can extract possible action groups. This problem is similar to finding a frequent *itemset* problem [9].

In a database composed of a large number of itemsets, if a sub-itemset (subset of an itemset) is contained in more than a specified threshold (called min support) of itemsets, it is considered *frequent*. The number of occurrences of a sub-itemset A is the support of A . The itemset that contains A is called its supporting itemset. For example, for a given itemset database $D = \{\{a, b, c, d, e\}, \{a, b, d, e, f\}, \{a, b, d, g\}, \{a, c, h, i\}\}$. The support of sub-itemset $\{a, b, d\}$ is 3, and its supporting itemsets are $\{a, b, c, d, e\}, \{a, b, d, e, f\}$ and $\{a, b, d, g\}$. If the

threshold of support is set as 3, the frequent sub-itemsets for D are $\{a\}:4$, $\{b\}:3$, $\{d\}:3$, $\{a,b\}:3$, $\{a,d\}:3$, $\{b,d\}:3$ and $\{a,b,d\}:3$, where the numbers are the supports of the corresponding sub-itemsets.

To solve the frequent itemset mining problem, many algorithms have been proposed. In this paper, we use an FP-tree-based mining algorithm called FPclose [9], which is one of the most efficient frequent itemset mining algorithms. In the example above, the frequent sub-itemsets $\{b\}$, $\{d\}$, $\{a,b\}$, $\{a,d\}$ and $\{b,d\}$ are not treated as itemsets since their supports are the same as their super-itemset $\{a,b,d\}$. FPclose only generates the sub-itemsets $\{a\}:4$ and $\{a,b,d\}:3$ as results. This significantly improves time and space performance as it avoids generating an exponential number of frequent sub-itemsets.

However, it is not sufficient to know only the possible action groups and their support values. It is also important to know in which transitions or states the extracted action groups occur. Unfortunately, the original algorithm FPclose algorithm only outputs the support values for each discovered group. Hence, we extend the FPclose algorithm to not only extracts possible action groups, but also identify the traces where these groups are used. The extended FPclose algorithm is described in Algorithm 3.

Algorithm 3 MINING ACTION GROUPS

Input: An instantiated Yakindu statechart $Y = \langle name, V, T, S \rangle$

Output: A set $G = \{g | g = \langle A, S_u, T_u, support \rangle\}$, where A is an actionset, S_u is a set of states in which A is used, T_u is a set of transitions in which A is used. *support* indicates how many times A occurs in Y .

- 1: Declare $\mathcal{A} = \emptyset$.
 - 2: **for** each state $s_i \in S$, and each transition $t_j \in T$ **do**
 - 3: $\mathcal{A} = \mathcal{A} \cup \{s_i.A_s^i\}$, and $\mathcal{A} = \mathcal{A} \cup \{t_j.A_t^j\}$.
 - 4: **end for**
 - 5: Use FPclose to mine the most frequent actionset from $\mathcal{A}$:
 $G \leftarrow FPCLOSE(\mathcal{A}, 1)$
 - 6: **for** each action group $g_i = \langle A_i, S_u^i, T_u^i, support_i \rangle \in G$ **do**
 - 7: Find states and transitions of Y where A_i is used, and put the states to S_u^i and put the transitions to T_u^i .
 - 8: **end for**
 - 9: **return** G
-

Table 1 shows the frequent action groupsthat are extracted from the Yakindu statechart model given in Fig. 1. Based on the result in Table 1, we can define a specified threshold number to narrow down the extracted frequent action groups. In addition, the *IAFinder* also can highlight states and transitions where the action group $\{raise\ BGI.PatientHasMetabolicAcidosis, metabolic_acidosis_hold_timer = 0\}$ occurs. Next section, we use a simplified cardiac arrest model to evaluate the *IAFinder*.

action groups	support
$\{raise\ BGI.PatientHasMetabolicAcidosis, metabolic_acidosis_hold_timer = 0\}$	2
$\{raise\ BGI.PatientHasRespiratoryAcidosis, metabolic_acidosis_hold_timer = 50\}$	1
$\{raise\ BGI.PatientHasRespiratoryAcidosis, respiratory_acidosis_hold_timer = 0\}$	1
$\{raise\ BGI.PatientHasHypercapnia, hypercapnia_hold_timer = 0\}$	2

Table 1: Example of Action Groups Presented Alg. 3

5 A CASE STUDY — USING THE *IAFINDER* TO IDENTIFY IMPLICIT ASSUMPTIONS FROM CARDIAC ARREST STATECHART MODEL

Cardiac arrest is the abrupt loss of heart function and can lead to patient death within minutes. American Heart Association (AHA) provided resuscitation guidelines for the urgent treatment of cardiac arrest [4]. In this section, we take a cardiac arrest assist system model implemented by Yakindu statechart to illustrate how to identify implicit assumptions with *IAFinder*. The completed statechart model of the cardiac arrest assist system can be found in [21].

With *IAFinder*, we extract 13 implicit constant variables out of total 45 variables, given in Table 2. For each con-

Extracted Constant Variables
<i>Kidney.BUN_High_Threshold, Kidney.Creatinine_High_Threshold, Kidney.Potassium_High_Threshold, BGI.pH_Low_Threshold, BGI.paCO2_High_Threshold, Physician.Deviation_Short_Timer, Physician.Deviation_Long_Timer, hypercapnia_jump_timer, metabolic_acidosis_deviation_counter, hypercapnia_deviation_counter, respiratory_acidosis_deviation_counter, respiratory_acidosis_jump_timer, normal_pH_deviation_counter</i>

Table 2: Extracted Constants from the Cardiac Arrest Assist System Statechart

stant variable, domain experts should explicitly specify the associated assumptions which supporting the initialization of the constant. For example, the implicit assumption behind $BGI.paCO2_High_Threshold = 7.5$ is that the initial value 7.5 is determined only for adult patients. For pediatric or elder patients, $BGI.paCO2_High_Threshold = 7.5$ does not accurately reflect the current patient situations. In addition, with the trace where the variable is used, domain experts can validate the impacts when the initial value is changed.

To extract possible action groups, we set the threshold value of *support* be 4. Table 3 shows the possible action groups whose support value is greater than or equal to 4.

For the action group $\{raise\ Arrhythmia.PatientIsSinus, sinus_deviation_counter=0\}$, the implicit assumptions behind this action group is that when a patient is admitted

Extracted Action Groups
$\{raise\ Arrhythmia.PatientIsPEA, PEA_deviation_counter=0\},$ $\{raise\ Arrhythmia.PatientIsVtach, vtach_deviation_counter=0\},$ $\{raise\ Arrhythmia.PatientIsSinus, sinus_deviation_counter=0\}$

Table 3: Extracted High-Frequency Action Groups from the Cardiac Arrest Assist System Statechart

to the procedure of Sinus Examination, a timer is triggered to record the diagnosis time, and when the diagnosis is confirmed, the timer should reset to zero. In addition, we also find that there exist some very low-frequency action groups which have interaction with high-frequency action groups. For example, we observe that $\{raise\ Arrhythmia.PatientIsSinus, vtach_deviation_counter=0\}$ occurs once, and the action group has similar structures with $\{raise\ Arrhythmia.PatientIsSinus, sinus_deviation_counter=0\}$ or $\{raise\ Arrhythmia.PatientIsVtach, vtach_deviation_counter=0\}$. Whether the low-frequency action groups designed on purpose or just a copy-pasting error raise another interesting problem that may be addressed by detecting violations based on high-frequency action groups.

Due to page limits, we do not list all the implicit assumptions that have been identified based on the extracted constant variables and action groups. Our results show that a large number of implicit, undocumented assumptions can be effectively identified based on extracted constant variables and high-frequency action groups by *IAFinder*.

6 CONCLUSION

This paper presents a tool called *IAFinder* that uses data mining techniques to efficiently and automatically extract potential implicit assumptions in the form of constant variables and high-frequency action groups from Yakindu statechart models. In addition, *IAFinder* also provides traces where constant variables and action groups occur in statechart models. We evaluate *IAFinder* with a cardiac arrest assist system statechart model. The results of the evaluation indicate that *IAFinder* can help to identify implicit assumptions and facilitate validation for domain experts. In addition, *IAFinder* can also help to detect potential errors by comparing low-frequency action groups with high-frequency action groups.

As *IAFinder* also provides traces where a constant variable or an action group occurs, we can potentially use the information to perform impact analyze on element changes, and identify what needs to be modified to realize an element change. This will be our future work. In addition, this paper only associates potential implicit assumptions with invariants in the forms of constants and high-frequency action groups, there may be other forms of invariants that implicit assumptions are associated with. We will investigate this in the future work as well.

REFERENCES

- [1] 2007. State Chart XML (SCXML): State Machine Notation for Control Abstraction. (February 2007). <http://www.w3.org/TR/scxml/#join>
- [2] Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183 – 235. [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
- [3] Tirumala A.S. 2006. An Assumptions Management Framework for Systems Software. In *Doctoral Thesis. University of Illinois at Urbana-Champaign*.
- [4] American Heart Association. 2017. About Cardiac Arrest. http://www.heart.org/HEARTORG/Conditions/More/CardiacArrest/About-Cardiac-Arrest_UCM_307905_Article.jsp. (2017).
- [5] Food, Drug Administration Center for Devices, Radiological Health, Office of Compliance Division of Analysis, and Program Operations. 2013. Medical Device Recall Report FY2003 to FY2012. <https://www.fda.gov/downloads/aboutfda/centersoffices/officeofmedicalproductsandtobacco/cdrh/cdrhtransparency/ucm388442.pdf>. (2013).
- [6] U.S. Food and Drug Administration. 2013. Class 1 Device Recall The HAMILTON T1. <https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRes/res.cfm?ID=115383>. (2013).
- [7] U.S. Food and Drug Administration. 2017. Medical Device Databases. <http://www.fda.gov/medicaldevices/deviceregulationandguidance/databases/>. (2017).
- [8] Zhicheng Fu, Chunhui Guo, Zhenyu Zhang, Shangping Ren, Yu Jiang, and Lui Sha. 2017. Study of Software-Related Causes in the FDA Medical Device Recalls. In *the 22nd International Conference on Engineering of Complex Computer Systems (ICECCS), 2017*.
- [9] Grahne Gosta and Zhu Jianfei. 2003. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations*.
- [10] Chunhui Guo, Shangping Ren, Yu Jiang, Po-Liang Wu, Lui Sha, and Richard Berlin. 2016. Transforming Medical Best Practice Guidelines to Executable and Verifiable Statechart Models. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPs)*. 1–10. <https://doi.org/10.1109/ICCPs.2016.7479121>
- [11] Ariane 501 inquiry board. 1996. Ariane 5 - Flight 501 failure report. <http://www.di.unito.it/~damiani/ariane5rep.html>. (1996).
- [12] Itemis. 2016. Yakindu Statechart Tools (SCT). <https://www.itemis.com/en/yakindu/statechart-tools/>. (2016). Retrieved July 31, 2016 from <https://www.itemis.com/en/yakindu/statechart-tools/>
- [13] Yu Jiang and etl. 2016. Data-centered runtime verification of wireless medical cyber-physical system. *IEEE Transactions on Industrial Informatics* (2016).
- [14] Meir M. Lehman and Juan F. Ramil. 2001. Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering* 11, 1 (01 Nov 2001), 15–44. <https://doi.org/10.1023/A:1012535017876>
- [15] S. Leue. 1999. Baby death due to software-controlled air bag deactivation?. In *ACM Risks Digest*.
- [16] Lewis, T. A. G., Mahatham, and Wrage L. 2004. Assumptions Management in Software Development. In *Technical Report. CMU*.
- [17] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 306–315. <https://doi.org/10.1145/1081706.1081755>
- [18] The W3C Document Object Model. 2017. Java DOM Parser - Overview. https://www.tutorialspoint.com/java_xml/java_dom_parser.htm. (2017).
- [19] Cleve B. Moler. 2008. Introduction to MATLAB / SIMULINK. In *Numerical Computing with MATLAB, Revised Reprint*. SIAM, Philadelphia, Chapter 1, 1–51. <https://doi.org/10.1137/1.9780898717952.ch1>
- [20] C. Adam Petri and W. Reisig. 2008. Petri net. *Scholarpedia* 3, 4 (2008), 6477. <https://doi.org/10.4249/scholarpedia.6477> revision #91646.
- [21] Maryam Rahmaniheris, Po-Liang Wu, Lui Sha, and Richard R. Berlin. 2016. An Organ-Centric Best Practice Assist System for Acute Care. In *29th IEEE International Symposium on Computer-Based Medical Systems, CBMS 2016, Belfast, UK and Dublin, Ireland, June 20-24, 2016*. 100–105. <https://doi.org/10.1109/CBMS.2016.12>
- [22] Lui Sha and Jose Meseguer. 2010. Analytical System Composition. In *The First Analytic Virtual Integration of Cyber-Physical Systems Workshop*.
- [23] Wikipedia. 2017. Invariant (computer science). (2017).