Iranian Journal of Science & Technology, Transaction B: Engineering, Vol. 34, No. B3, pp 235-256
Printed in The Islamic Republic of Iran, 2010

SYNTHESIZING A SPECIFICATION-BASED MONITOR FOR SAFETY REQUIREMENTS*

S. M. BABAMIR[1**] AND S. JALILI[2]

[1]Dept. of Computer Engineering, University of Kashan, Kashan, I. R. of Iran
Email: babamir@kashanu.ac.ir
[2]School of Electrical and Computer Engineering, Tarbiat Modares University, Tehran, I. R. of Iran

Abstract–Safety-critical systems such as medical and avionic ones are the systems in which failure to satisfy the user requirements may put man's life and resources in jeopardy. Since the adequate reliability of the software of such systems may be unobtainable via formal methods and the software testing approach single-handedly, *verification of run-time behavior* of software against user requirements violation is considered as a complementary approach. However, the synthesis of such a run-time verifier, hereafter we have called it a *monitor*, is confronted with the challenging problem of verifying low-level run-time behavior of target software against high-level user requirements violation. To solve this problem, we propose an approach in two phases. In the first phase, we obtain user requirements and then specify their *violation* formally. This formal specification is a high-level version of user requirements violations and should be mapped to a low-level one. To this end, in the second phase we extract a tabular automaton from the formal specification of user requirements violations in order to determine a state-based specification of the violations. This low-level specification, which constitutes the core of the monitor, determines those states which target software should not reach. To show the effectiveness of our approach, we apply it to the synthesis of a monitor for verifying behavior of the *Continuous Insulin Infusion Pump* (CIIP) system.

Keywords– Safety-critical systems, run-time verification, event-based specification, state-based specification

1. INTRODUCTION

Safety-critical systems such as medical, avionic, nuclear reactors and chemical plants are systems that, when they fail, may result in loss of human life or damage to the environment [1, 2, 3]. NASA states, "Software is considered safety-critical if it controls or monitors hazardous or safety-critical hardware or software" [4]. On the other hand, as software becomes an important component of safety-critical systems its size and complexity increases too. This leads to a high probability of the system fault. Therefore, the major aim of safety-critical software development is the development of reliable systems in spite of their complexity. Moreover, the verification and validation process plays a vital role in software development because it is required to ensure that software behaves as intended. The aims of software verification and validation are: (1) exposing the faults of the specification, design, and implementation of the software, (2) assuring software reliability and (3) reducing costs by preventing software failures.

Although use of static verification and testing approaches to detect software faults are necessary, they may be inadequate. Using formal methods for static verification is subject to (1) undecidability and difficulty of proving some properties of specifications, and (2) unpredictability of the run-time environment at the specification time. This is why Leucker and Schallhart state that while model checking

*Received by the editors May 24, 2008; Accepted June 7, 2010.
**Corresponding author

verification is subject to the *infinity* of the state space of software *runs*, run-time verification faces a finite *execution* of software [5]. Rosu et al showed an experimental case that was generally undecidable, but in run-time verification it was decidable [6]. They state, "We show that although verification of memory safety is in general undecidable, even when restricted to closed, terminating programs, runtime verification of strong memory safety is a decision procedure for this class of programs". Using software testing is subject to (1) impracticality of testing software by all possible input values (that is why Hoare, Dijkstra [7] and Parnas [8] believe: "Non-exhaustive testing can only show the presence of errors not their absence") and (2) difficulty of selection of a proper subset of test data.

Safety-critical software should be verified properly in order to achieve greater reliability because: (1) there may be some faults left despite verifying specification and testing software, (2) implemented software may be inconsistent with its specification, and (3) the environment in which software will run may be unpredictable. Safety-critical software must be bound to 10^{-9} failures/hour or less [9], while the reliability growth models have proven that software testing can guarantee 10^{-4} failures/hour in the best case [10]. In addition, cases such as: (1) codes that a compiler adds to the object code of software, (2) whether a compiler correctly generates an executable code [8], and (3) transient errors such as data corruption and hardware malfunctions are other aspects of the problem. Iyer and Verlardi state that around 10% of software errors and 35% of hardware errors are transient errors, which could not be identified in advance [11].

The above-mentioned problems justify *verification of run-time behavior* of software against user requirements violation. However, the synthesis of such a run-time verifier, called a *monitor* is confronted with the challenging problem of verifying low-level run-time behavior of target software against high-level user requirements because they are not analogous. Since such a monitor is a run-time verifier with a core synthesized based on the specification of high-level user requirements, we call it a specification-based monitor.

To synthesize the monitor, we propose an approach in two phases. In the first phase we obtain user requirements from the problem domain, and then specify their *violation* of them formally. This phase includes two steps. In the first step, we obtain user requirements from the problem domain via a specific method. In the second step, we represent an event-based specification of violation of user requirements based on interactions between the system and its environment in *Event Calculus* (EC). EC [12-13] makes three contributions to our approach: (1) it facilitates the specification of the violation of user requirements through a logical and *temporal* method, (2) it facilitates mapping event-based specification into a state-based one, and (3) it is well-suited to specify user requirements of the safety-critical systems which control and manage their environment based on environment events. The second phase includes two steps as well. In the first step, we extract a tabular automaton from the event-based specification of user requirements violation in order to determine a state-based specification. This specification, which constitutes the core of the monitor, determines those states which target software should not enter. In the second step, we deal with designing the monitor via the *State Design Pattern* [14], which is a method for designing an automaton. As stated, each phase includes two steps; so we refer to them as steps 1 to 4 where the second and third steps represent two views respectively, event-based and state-based, of a system behavior deviation. Presenting these two views together implies two sides of safety-critical systems behavior: 1) while they are event-oriented in interaction with their environment, 2) are state-based within themselves.

We explain the approach in Section 2 and its steps in Sections 3 to 6. In Section 7, in order to show the effectiveness of the approach, we apply each of the steps to derive the monitor of a safety-critical system called Continuous Insulin Infusion Pump (CIIP) [15] from the system problem domain. Finally, in Section 8, we draw some conclusions and we state the features of the approach in comparison with other related approaches.

2. PRELIMINARIES

The run-time verification of software behavior is a lightweight formal method in which no theorem proving method is applied, modeling is restricted; accordingly, mathematics embodiments are reduced. This method uses features of the specification verification and the software testing approaches but unlike them, it just addresses verification of the *current execution* of software. An execution is a *finite sequence* of observed program states $\sigma=s_1s_2\ldots s_n$ where $|\sigma|=n$ is the length of the execution trace [16]. Leucker and Schallhart state, "Runtime verification deals only with observed executions as they are generated by the real system. An execution of a system is a finite prefix of a run and, formally, it is a finite trace. When running a program, we can only observe executions, which, however, restrict the corresponding evolving run as being their prefix" [17].

This is why the approach does not involve: (1) the complexity of theorem proving methods and state explosion problem of model checking methods and (2) the problem of generating appropriate and sufficient test data to execute software because run-time verification just considers software execution by providing real data originated from run-time environment. The run-time verification approach has shown its effectiveness in comparison with other verification methods [18-20].

Figure 1 shows the architecture of a monitor [21] where the state of executing software changes in response to an environment event. The *observer* receives the software state and sends it to the *analyzer* if the state is a concern. Then, the *analyzer* verifies software states against properties of requirements and gives the result to the event handler. If the current execution of software violates a requirement property, the monitor will react to it through the *event handler*. In our approach, we assume that: (1) software is a white-box and the observer is a part of the executing software that announces states of interest to the analyzer and (2) the event handler is a part of the analyzer where reaction is a warning message. Therefore, hereafter we refer to the analyzer and event handler in Fig. 1 as the *monitor*.

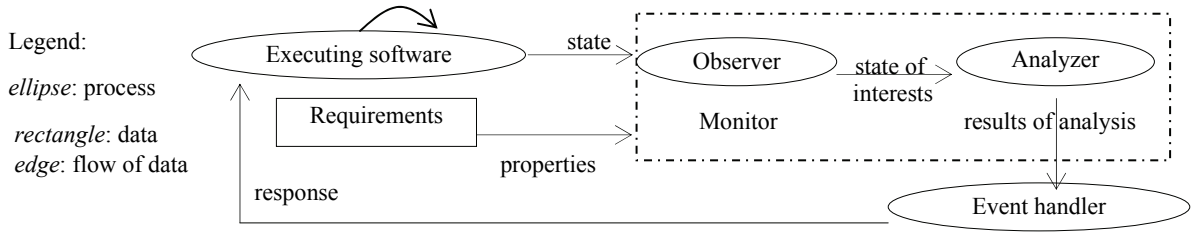

Fig. 1. Architecture of a monitor [21]

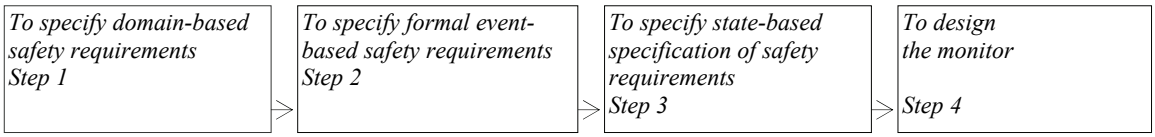

Fig. 2. The proposed approach to derive a monitor from safety requirements

Figure 2 shows the four steps of our approach to synthesize the *monitor* (i.e., analyzer) which is in charge of current software behavior against violation of user requirements. The user requirements of safety-critical systems are stated as *safety requirements*. According to [22], a *safety requirement* asserts, "Nothing bad happens" where the bad event is a violation of a user requirement. Therefore, to monitor a

safety-critical system against safety requirements, run-time behavior of its software should be verified against the user requirements violation. In a medical system, for instance, "Normal pressure" is a user requirement and "Low/high pressure" is a violation of the user requirement. Therefore, the user requirement, "Normal pressure" can be proposed in the form of a safety requirement, "No low or No high pressure should happen" where the low/high pressure is a violation of the user requirement "Normal pressure".

In Step 1, in keeping with the view of the problem domain, we specify safety requirements through determining the violation of user requirements. In Step 2, we present a formal and event-based specification of safety requirements because safety-critical systems are mostly used to control and manage their environment based on the environment events. In Step 3, we consider the event-based specification of the safety requirements violation and derive its state-based specification version. Therefore, the state-based specification shows those states that target software should not enter. In fact, Step 3 is a significant step by which we address the problem of deriving violating low-level run-time behavior from the violation of high-level specifications. In Step 4, we design the monitor based on patterns where each pattern is the state-based specification of a violation.

Altogether we present three levels of specification and derive each level from the previous one: (1) domain-based specification, which is a user level one, (2) event-based specification, which is formal and event-based, and (3) state-based specification, which is a low-level one. Accordingly, whenever an environment event happens, the monitor can compare software behavior with the state-based specification of safety requirements violation. Our approach is a systematic one in which the first step is fulfilled manually but the rest of the steps are fulfilled automatically. In the following four sections, we explain the steps of the proposed approach.

3. SPECIFYING DOMAIN-BASED SAFETY REQUIREMENTS (STEP 1)

A safety-critical system such as diabetes control system aims at managing a safety-critical environment such as a diabetic where the environment has some requirements. If such a system fails to satisfy the requirements of its environment, the environment may face some disasters. For example, if a diabetes control system fails to satisfy requirement "Normal blood sugar", the diabetic may be afflicted with cerebral, eye, heart, or kidney diseases. As we stated in Section 2, user requirements of safety-critical systems are stated as safety requirements. In the diabetes system, for instance, "Diabetic's blood sugar should be normal" is a user requirement that can be stated in the safety requirement, "Diabetic's blood sugar should not be low or high". Therefore, to monitor target software its run-time behavior for the violation of the user requirements should be verified.

To specify safety requirements, we use problem domain and determine *violation* of user requirements. Since the problem domain is stated in terms of vocabulary of the system user, it is called domain-based specification. Inspired by [23], in order to determine the violation of the domain-level requirements more naturally and clearly, first we designate an environment by its *attributes* where an attribute is an important characteristic of the system environment. Afterwards, we define *events* and *invariants* in terms of the attributes. Finally, we specify the violation of the user requirements as an event together with some invariants, if any. An event indicates a crucial change in an environment attribute value, but an invariant indicates no crucial change in another attribute value while an event happens.

In the diabetes control system, for instance, the diabetic is the system environment where "A sharp fall in his/her blood sugar", indicated by $normal \rightarrow low$, is the event that violates the user requirement "Normal blood sugar". However, "A rise in his/her blood sugar", indicated by $normal \rightarrow high$, is the event

which violates the user requirement "Normal blood sugar" if the invariant "Insufficient insulin in the system reservoir" holds. In order to define events and invariants formally, we first define data abstraction.

a) Data abstraction

The system environment is specified by some attributes where an attribute, indicated by Att, is a *well-ordered* set of Boolean, integer or real values. The order property distinguishes the Boolean set {true, false} (with the well-ordering specified by the order of writing its elements) from the Boolean set {false, true}. A *well-ordered* set is a totally (linearly) ordered set if the set and every one of its subsets contains a first element [24]. For example, the set of integers Z={…-2,-1,0,1,2,…} is a linearly ordered set but not a well-ordered one because it does not have a first element. However, the set of positive integers P={1,2,3,…} is a well-ordered set because P and every one of its subsets has a first element. The *order* relation "≤" (i.e., "less than or equal") is a relation on any set Att satisfying the following three properties P1, P2, and P3.

P1. Reflexive: for all $a \in Att$, $a \leq a$,
P2. Antisymmetric: for all $a, b \in Att$, if $a \leq b$ and $b \leq a$ then $a=b$,
P3. Transitive: for all $a, b,$ and $c \in Att$, if $a \leq b$ and $b \leq c$ then $a \leq c$.

Definition. A set is said to be bounded if and only if it is a subset of a finite interval. For example, the set $\{ 1/n : n \geq 1 \}$ is bounded because it is certainly a subset of the closed unit interval [0,1] [24].

Let a and b be two distinct integer or real numbers where $a < b$. An interval with endpoints a and b is defined as a closed-open interval from a to b, i.e., [a,b). Now, based on the problem domain an Att is split up into *bounded* and disjoint intervals with the order relation "≤" such that for two consecutive intervals, say, [a,b) and [b, c), the supremum of the first interval is the minimum of the second one. In the case of integer numbers, such bounded intervals are finite. However, in the case of real numbers they may be infinite and uncountable. This is not an obstacle to our approach because just the endpoints of each interval are our concerns. Therefore, in the case of real numbers, without loss of generality, we can split an *Att* up into closed-open intervals such that each interval consists of a minimum and a supremum. For instance, suppose that *velocity* is the environment attribute whose value reaches 10.0 mph at most and 3.0, 7.5, and 10 are its boundary values. Based on these domain values, we split the velocity up into intervals [0…3.0), [3.0…7.5), and [7.5…10.0]. When an input value is received from the system environment, we compare it with the minimum value of each interval to determine its corresponding interval. Now we deal with the defining properties of intervals on the integer or real attributes:

$Att_p = \{a_0 , a_1 , \ldots , a_n\} \equiv \bigcup_{i=1}^{k} I_{i,p}$ // Att_p denotes attribute *p* and it is a well-ordered set. $I_{i,p}$ is the i^{th} closed-open interval of Att_p (1)

$I_{1,p} = [a_0 , \ldots , a_j), \ldots , I_{k,p} = [a_m , \ldots , a_n)$: $\forall I_{i,p} \subset Real \vee \forall I_{i,p} \subset Integer$, $i \in [1..k]$ and p is an attribute (2)

// each $I_{i,p}$ is an interval on the Att_p whose minimum and supremum are determined based on the problem domain

$\forall i: I_{i,p} \neq \emptyset$ // each interval of an attribute is a non-empty set (3)

$I_{i,p} \cap I_{j,p} = \emptyset : i \neq j, I_{i,p}, I_{j,p} \in Att_p$ // intervals of an attribute are disjoint (4)

$a < b \{ \begin{matrix} a \in I_{i,p}, b \in I_{j,p}, i<j \\ a,b \in I_{i,p}, a < b \text{ in } I_{i,p} \end{matrix}$ // because an attribute is a well-ordered set; so each of its subset is a well-ordered one [24] (5)

Having determined intervals, we get down to defining *interval indicators*. Suppose $I_{i,p}$ is the i^{th} interval of Att_p and τ is a sequence number indicating an ordering on the measured values of attribute Att_p. An *Interval Indicator*, indicated by $(II_{i,p})_\tau$, is a mapping for interval $I_{i,p}$ at time τ in the form: $(II_{i,p})_\tau$: $(Att_p)_\tau$

$\rightarrow \begin{cases} \text{true} | (Att_p)_\tau \in I_{i,p} \\ \text{false} | \text{otherwise} \end{cases}$. Hereafter, we call each measuring time of an attribute as an *occurrence*. Therefore, we say, $(II_{i,p})_\tau$ holds if the τ^{th} *occurrence* of Att_p belongs to $I_{i,p}$. In the diabetes system, for example, suppose that Att_1 indicates the blood sugar attribute; accordingly $I_{2,1}$ will indicate the second interval of Att_1 and expression "$(II_{2,1})_3$ holds" means, "The third occurrence of the blood sugar belongs to the second interval of Att_1". It should be noticed that at any time, just one interval indicator of an attribute may hold because each value of Att_p belongs to just one of its intervals (Relation (6)).

$$[(II_{i,p})_\tau \wedge (II_{j,p})_\tau] : i \neq j, \forall (I_{i,p}, I_{j,p}) \in Att_p, \text{ where } Att_p \text{ is an attribute} \quad (6)$$

b) Event definition

We say an event has happened if a change in the value of an attribute such as Att_p has caused Att_p to change from its $I_{i,p}$ to the adjacent interval, $I_{(i-1),p}$ or $I_{(i+1),p}$ or vice versa. Accordingly, we have assumed that there would be no sharp change in the value of Att_p. Now, we formally define an event.

If two consecutive *occurrences* of Att_p, indicated by τ^{th} and $(\tau+1)^{th}$ belong to two adjacent intervals of Att_p, we say an *event* has happened at time τ. This means that an event has happened when: (1) the τ^{th} measured value belongs to $I_{(i-1),p}$ or $I_{(i+1),p}$ and the $(\tau+1)^{th}$ measured value belongs to $I_{i,p}$ or (2) vice versa. The former case is indicated by Relation (7) and the latter case by Relation (8). Relation (7) states, "τ is the last time that Att_p has been in $I_{(i-1),p}$ or $I_{(i+1),p}$ and at this time a change in the value of Att_p has caused Att_p to leave $I_{(i-1),p}$ or $I_{(i+1),p}$ and it then enters $I_{i,p}$ at $\tau+1$. Similarly, Relation (8) states, "τ is the last time that Att_p has been in $I_{i,p}$ and at this time a change in the value of Att_p has caused Att_p to leave $I_{i,p}$ and enters $I_{(i-1),p}$ or $I_{(i+1),p}$ at $\tau+1$. As Relation (7)/(8) shows, we have used just $II_{i,p}$. This means that our concern has been just *entering*/*leaving* $I_{i,p}$ not leaving/entering its adjacent interval.

$$(Event_{i,p})_\tau \stackrel{def}{\equiv} [\sim(II_{i,p})_\tau \wedge (II_{i,p})_{\tau+1}] \quad \text{// } \textit{entering} \text{ interval } I_{i,p} \text{ at } \tau+1 \quad (7)$$

$$(Event_{i,p})_\tau \stackrel{def}{\equiv} [(II_{i,p})_\tau \wedge \sim(II_{i,p})_{\tau+1}] \quad \text{// } \textit{leaving} \text{ interval } I_{i,p} \text{ at } \tau+1 \quad (8)$$

In the case of Boolean (i.e., $Att_1 = \{\text{true, false}\}$), we have a partition, $I_{1,1}=\{\text{true, false}\}$ and just an interval indictor, $II_{1,1}$ is adequate. Therefore, there are just two events:

$$(Event_{1,1})_\tau \stackrel{def}{\equiv} [\sim(II_{1,1})_\tau \wedge (II_{1,1})_{\tau+1}] \quad \text{// entering interval } I_{1,1} \text{ at } \tau+1 \quad (9)$$

$$(Event_{1,1})_\tau \stackrel{def}{\equiv} [(II_{1,1})_\tau \wedge \sim(II_{1,1})_{\tau+1}] \quad \text{// leaving interval } I_{1,1} \text{ at } \tau+1 \quad (10)$$

where, Relations (9) and (10) can be subsumed by Relations (7) and (8).

Note that events are mutually exclusive, i.e., there are *no concurrent events*. In other words, we assume that at most an event happens at each time step. This means that at the value of at most, an attribute changes at each time step (Relations 11 to 14). The left hand sides of Relations 11 to 14 indicate the change of value of Att_p at τ and the right hand sides of the relations indicate that there may not be a change in the value of another attribute, say Att_q, at the same time. In addition, as we stated in Relation (6), at most, a value of an attribute may change at each time step.

$$\text{if } [\sim(II_{i,p})_\tau \wedge (II_{i,p})_{\tau+1}] \wedge \sim(II_{k,q})_\tau \text{ then } \sim(II_{k,q})_{\tau+1} \quad (11)$$

$$\text{if } [\sim(II_{i,p})_\tau \wedge (II_{i,p})_{\tau+1}] \wedge (II_{k,q})_\tau \text{ then } (II_{k,q})_{\tau+1} \quad (12)$$

$$\text{if } [(II_{i,p})_\tau \wedge \sim(II_{i,p})_{\tau+1}] \wedge \sim(II_{k,q})_\tau \text{ then } \sim(II_{k,q})_{\tau+1} \quad (13)$$

$$\text{if } [(II_{i,p})_\tau \wedge \sim(II_{i,p})_{\tau+1}] \wedge (II_{k,q})_\tau \text{ then } (II_{k,q})_{\tau+1} \quad (14)$$

where $I_{i,p} \in Att_p$, $I_{k,q} \in Att_q$, $p \neq q$

c) Invariant definition

Given two consecutive *occurrences* of Att_q, indicated by τ^{th} and $(\tau+1)^{th}$, we say there is an invariant on $I_{j,q}$ if: (1) both occurrences belong to $I_{j,q}$ or (2) neither of them belongs to $I_{j,q}$. We call the former case a *true invariant* shown as Relation 15 ($II_{j,q}$ *holds*) and the latter one a *false invariant* shown as Relation 16 ($II_{j,q}$ *does not hold*). Unlike events, invariants may be concurrent. This means that a combination of true and false invariants may hold at the same time provided each invariant belongs to just a distinct attribute. We show the coexistence of several invariants, i.e. concurrent invariants by $\circ_{i,r}(In_{i,r})_\tau$; this specifies some invariants on intervals of *separate* attributes; so, for every two concurrent invariants, "$(In_{i,p})_\tau$" and "$(In_{j,q})_\tau$", we have $p \neq q$.

$$(In_{j,q})_\tau \stackrel{def}{\equiv} [(II_{j,q})_\tau \wedge (II_{j,q})_{\tau+1}] \quad \text{// true invariant means, "} II_{j,q} \text{ has been true at } \tau \text{ and } \tau+1" \quad (15)$$

$$(In_{j,q})_\tau \stackrel{def}{\equiv} [\sim(II_{j,q})_\tau \wedge \sim(II_{j,q})_{\tau+1}] \quad \text{// false invariant means, "} II_{j,q} \text{ has been false at } \tau \text{ and } \tau+1" \quad (16)$$

d) Domain-based safety requirement

As we stated in Sections 2 and 3, a safety requirement indicates that a bad event must not happen where the bad event indicates a user requirement violation. We show the violation as "$(Event_{i,p})_\tau \wedge \circ_{i,r}(In_{i,r})_\tau$", consisting of an event (Relation 7 and Relation 8) and zero or some true/false invariants (15 or 16). In the diabetes system, suppose that Att_1 indicates "blood sugar" whose third interval is the high interval; so, $II_{3,1}$ indicates, "High blood sugar" and suppose that Att_5 indicates "insulin dose" whose first interval is the insufficient interval; so, $II_{1,5}$ indicates, "Insufficient insulin dose". Accordingly, "$(Event_{3,1})_\tau \wedge (In_{1,5})_\tau$" where $(Event_{3,1})_\tau$ is specified as "$[\sim(II_{3,1})_\tau \wedge (II_{3,1})_{\tau+1}]$", and $(In_{1,5})_\tau$ is specified as "$[(II_{1,5})_\tau \wedge (II_{1,5})_{\tau+1}]$" shows a violation. Invariant "$(In_{1,5})_\tau$" states that at the time of the event, the system reservoir has contained insufficient insulin.

In the case of no invariant, a safety requirement will be specified as $\sim(Event_{i,p})_\tau$. In the diabetes system, for example, the statement, "The blood sugar *must not* fall below normal" indicated by $\sim(Event_{1,1})_\tau$ is a safety requirement. According to Relations (11) to (14), we assume that there would be no concurrent events. This means that the system would face just one safety requirement at any given time and accordingly the monitor behavior is definite when it is verifying target software behavior in accordance with safety requirements.

4. SPECIFYING EVENT-BASED SAFETY REQUIREMENTS (STEP 2)

In Section 3, we specified the user requirements violation at the domain-level as events and invariants. This helped us to provide an abstract event-based specification of user requirements violation. In this section, we intend to map the specification into *EC* (Event Calculus) *Formulae*, providing a formal event-based and time-dependent specification.

EC was introduced by Kowalski and Sergot [12] to represent *actions* in time intervals. EC is based on three domains represented in Fig. 3 where a *fluent* is a logical variable or a predicate such as "the device is faulty" whose truth-value changes over time and actions, called events change the truth-value of a fluent [25]. A deduction system of EC takes action in terms of time and its effects as input and then produces some facts as output. The actions by *Happens*, the effects by *Initiates*, *Terminates* and *Clipped* and the facts by *HoldsAt* are specified. In this paper, we use the Simple EC called SEC [13] in which time points are used rather than time intervals. SEC consists of a main axiom S (Fig. 4) and some predicates (Table 1) [25] where α, β, and τ indicate an action, a fluent, and a time point, respectively. The axiom S states fluent β will be true at time τ if an action (such as α_1) happens so that the fluent β is initiated before τ and does not change by the time τ. In order to map the domain-based specifications of the user requirements

violation into SEC formulae, we deal with determining indicators and specifying invariants and events in SEC.

1- τ : time (with algebraic structure and linear ordering relations)
2- $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ $\therefore \varphi_i$ is a fluent
3- $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ $\therefore \alpha_i$ is an action

Fig. 3. Event Calculus domains

S Axiom:

$$\text{HoldsAt}(\beta, \tau) \leftarrow \text{Happens}(\alpha_1, \tau_1) \wedge \text{Initiates}(\alpha_1, \beta) \wedge \tau_1 < \tau \wedge \sim\text{Clipped}(\tau_1, \beta, \tau)$$

$$\text{Clipped}(\tau_1, \beta, \tau) \equiv \text{Happens}(\alpha_2, \tau_2) \wedge \text{Terminates}(\alpha_2, \beta) \wedge \tau_1 < \tau_2 \wedge \tau_2 < \tau$$

Fig. 4. The SEC main axiom S [13]

Table 1. SEC Predicates [25]

	Predicate	Meaning
1	Initiates(α,β)	fluent β starts to hold after action α
2	Terminates(α,β)	fluent β stops to hold after action α
3	$\tau_1 < \tau_2$	time point τ_1 is before time point τ_2
4	Happens(α,τ)	action α occurs at time τ
5	HoldsAt(β,τ)	fluent β holds at time τ
6	Clipped(τ_1,β,τ_2)	fluent β stops to hold between τ_1 and τ_2

a) Determining indicators in EC

We consider an interval indicator as a fluent in EC because both are logical variables. Therefore, an attribute consists of a number of fluents. As stated above, facts would be specified as the predicate HoldsAt(β, τ) where β is a fluent. Hence, we can show $\sim(II_{i,p})_\tau$ as the predicate "~HoldsAt($II_{i,p}$,τ)" and $(II_{i,p})_{\tau+1}$ as the predicate "HoldsAt($II_{i,p}$,τ+1)" in EC. The predicate "~HoldsAt($II_{i,p}$, τ)/ HoldsAt($II_{i,p}$, τ)" means that the fluent $II_{i,p}$, has not held/has held at τ^{th} occurrence of the Att_p.

b) Specifying events

According to EC, an event may be represented as its effect on the truth-value of a fluent [25] which corresponds to an interval indicator. We have used this notion throughout this paper. In Section 3.b, we defined an event as Relation (7) or (8 in interval indicators terms. According to Section 4.a, we can represent $\sim(II_{i,p})_\tau$ as the predicate "~HoldsAt($II_{i,p}$, τ)" and $(II_{i,p})_{\tau+1}$ as the predicate "HoldsAt($II_{i,p}$, τ+1)"; so, we can state Relations (7) and (8) as Relations (17) and (18) respectively. Relation (17) states, "The fluent $II_{i,p}$ has not held at the τ^{th} occurrence of Att_p, but it has started to hold at its $(\tau+1)^{th}$ occurrence. On the other hand, the main axiom of EC (Fig. 4) states, "A fluent will start to hold if an initiation action occurs". Therefore, we conclude from Relation (17) that an action has occurred at the τ^{th} occurrence of Att_p that starts the initiation of the fluent $II_{i,p}$ at the $(\tau+1)^{th}$ occurrence.

Now, we represent Relations (11) to (14), indicating no concurrent events, as Relations (19) to (22). These relations state that truth-value of just a fluent can change at any given time.

$$(Event_{i,p})_\tau \stackrel{def}{\equiv} [\sim\text{HoldsAt}(II_{i,p}, \tau) \wedge \text{HoldsAt}(II_{i,p}, \tau+1)] \quad // \; II_{i,p} \text{ has become true at } \tau+1 \qquad (17)$$

$$(Event_{i,p})_\tau \stackrel{def}{\equiv} [\text{HoldsAt}(II_{i,p}, \tau) \wedge \sim\text{HoldsAt}(II_{i,p}, \tau+1)] \quad // \; II_{i,p} \text{ has become false at } \tau+1 \qquad (18)$$

$$\text{if } [\sim\text{HoldsAt}(II_{i,p},\tau) \wedge \text{HoldsAt}(II_{i,p},\tau+1)] \wedge \sim\text{HoldsAt}(II_{k,q},\tau) \text{ then } \sim\text{HoldsAt}(II_{k,q},\tau+1) \qquad (19)$$

$$\text{if } [\sim\text{HoldsAt}(II_{i,p},\tau) \wedge \text{HoldsAt}(II_{i,p},\tau+1)] \wedge \text{HoldsAt}(II_{k,q},\tau) \text{ then } \text{HoldsAt}(II_{k,q},\tau+1) \qquad (20)$$

$$\text{if } [\text{HoldsAt}(II_{i,p},\tau) \wedge \sim\text{HoldsAt}(II_{i,p},\tau+1)] \wedge \sim\text{HoldsAt}(II_{k,q},\tau) \text{ then } \sim\text{HoldsAt}(II_{k,q},\tau+1) \qquad (21)$$

$$\text{if } [\text{HoldsAt}(II_{i,p},\tau) \wedge \sim\text{HoldsAt}(II_{i,p},\tau+1)] \wedge \text{HoldsAt}(II_{k,q},\tau) \text{ then } \text{HoldsAt}(II_{k,q},\tau+1) \qquad (22)$$

where $II_{i,p} \in Att_p$, $II_{k,q} \in Att_q$, $p \neq q$

c) Specifying invariants

In Section 3.c, we defined an invariant as Relations (15) and (16). According to Section 4.a, we can represent them by a combination of predicates "HoldsAt". Therefore, Relation (15) or (16) is stated as Relation (23) or (24), which state that the truth-value of fluent $II_{i,p}$ has not changed at two consecutive occurrences of Att_p.

$$(In_{j,p})_\tau \stackrel{def}{\equiv} [\text{HoldsAt}(II_{j,p}, \tau) \wedge \text{HoldsAt}(II_{j,p}, \tau+1)] \quad \text{// true invariant in EC} \quad (23)$$

$$(In_{j,p})_\tau \stackrel{def}{\equiv} [\sim\text{HoldsAt}(II_{j,p}, \tau) \wedge \sim\text{HoldsAt}(II_{j,p}, \tau+1)] \quad \text{// false invariant in EC} \quad (24)$$

d) Specifying safety requirements

As we stated in Sections 2 and 3, a safety requirement means that a bad event would not happen where the bad-event is a user requirement violation. A user requirement violation, indicated by "$(Event_{i,p})_\tau \wedge \circ_{i,r}(In_{i,r})_\tau$", is specified as an event (Relations 23 or 24) and zero or some invariants (Relations 23 or 24). Having specified the violation of a user requirement as "$(Event_{i,p})_\tau \wedge \circ_{i,r}(In_{i,r})_\tau$", we specify a safety requirement as "$(Event_{i,p})_\tau \wedge \sim\circ_{i,r}(In_{i,r})_\tau$", indicating a bad event will not happen. In the case of no invariant, we specify the safety requirement as $\sim(Event_{i,p})_\tau$.

5. SPECIFYING STATE-BASED SAFETY REQUIREMENTS (STEP 3)

In this section, we aim to present a method by which we map the event-based specification of violations, stated by EC formulae, to the corresponding state-based one. We are thinking of the state-based specification as a *tabular automaton* (state machine) and we *automatically* map predicates of EC formulae to the elements of the automaton. Shown in Table 2, the structure of the automaton is influenced by the transition table introduced by "Software Cost Reduction" method [26]. Before mapping the predicates to the elements of the automaton, we explain the syntax and semantic of the automaton.

An automaton consists of two distinct sets, state and transition where each state is either an intermediate state or a final one. We use "*mode*" instead of the state and inspired by [26], we think of a mode as a system state in which the system involves performing an *operation*. Consider the determination of the dose value in the diabetes system, for instance. The system software should perform the computation operation to determine the value of the dose attribute; so, the "Computing" mode is considered as an operation mode.

In Table 2, we have considered: (1) the intermediate modes as safe (normal) operation modes of the system shown by the "Current Mode" column, (2) the final modes as unsafe or critical modes shown by the "Violated Mode" column, and (3) transitions as violations shown by the "Event/Invariant" column. Such a transition moves the system from a normal mode into an unsafe/critical one. To specify the state-based specification of the requirements violation, we should map the event-based one, shown by "$(Event_{i,p})_\tau \wedge \circ_{i,r}(In_{i,r})_\tau$" in Section 4.d to the transitions of the automaton. In fact, each transition causes a requirement violation.

As we stated in Section 4.d, a user requirement violation is shown by an event with or without invariants. Therefore, in Table 2 we show the transitions of the automaton as the "Event/Invariant" column and we split the column up into attributes indicated by distinct sub-columns "Att_p, p=1..n" (n is the number of attributes). To show the fluents of each attribute, we split the Att_p column up into sub-columns indicated by $II_{i,p}$. In the "Event/Invariant" column, we show an event as "@T" or "@F" and an invariant as "t" or "f". "@T" means that the value of the fluent is false in the current mode and becomes true in the violated mode when the transition fires. Similarly, "@F" means that the value of the fluent is true in the current mode and false in the violated mode. "t" means that the value of the fluent is true in the

current mode and remains true when the automaton goes to the violated mode. Similarly, “f” means that the value of the fluent is false in both modes. “x” denotes that the value of the fluent plays no role in firing the transition. Firing a transition indicates the occurrence of a violation and consequently, changes to an unsafe/critical mode.

The proposed tabular automaton is *deterministic* because each transition of the automaton is derived from Relation 17 or 18 and according to Relations (19) to (22), events are mutual exclusive. This means that each transition would have just one "@F/@T"; however, there may be a number of invariants, indicated by "t" or "f", in which each t/f belongs to a distinct attribute. For example, consider the transition “@T f t x” under some fluents of Att_1, Att_2, Att_3, and Att_4, respectively. We realize that the system enters an unsafe/critical when, (1) the truth-value of the fluent II_{i1} changes from false to true (i.e. an event, denoted by "@T") and (2) the truth-values of the fluent $II_{j,2}$ and fluent $II_{k,3}$ stay false and true, respectively.

Such a specification which shows unsafe or critical (violation) modes, is used to verify run-time system operation modes. In addition, it builds a bridge over the gap between the environment events and the run-time system operation modes to verify run-time behavior of the system in accordance with high-level domain-based safety requirements.

To map the event-based specification of the user requirements violation, stated by the EC formulae into the state-based one, stated by tabular automaton, we have derived ingredients of the automaton consisting of “@T/@F”, “t/f” and “Enter mode $II_{i,p}$” from predicates of the EC formulae consisting of compound predicates in Table 3. Now, we deal with verifying the derivations.

$$[\sim HoldsAt(II_{i,p},\tau) \wedge \sim HoldsAt(II_{i,p},\tau+1)] \equiv "II_{i,p} = f" \quad (25)$$

On the one hand, according to the semantics of the automaton mentioned above, the right hand side of Relation (25) means that the value of $II_{i,p}$ is false in the current mode and remains false when the automaton goes to its next mode; so, it holds in neither current mode nor the next mode. On the other hand, based on Relation (24), the left part of Relation (25) shows a *false invariant* (Section 3.c). According to Section 4.c, this means that $II_{i,p}$ holds at neither the τ^{th} nor the $(\tau+1)^{th}$ occurrence of Att_p. If we think of the system operation as an attribute where each specific operation mode is an occurrence of a system operation, "τ" and "τ+1" will indicate two consecutive system operation modes. Accordingly, the left hand side of Relation (25) means that $II_{i,p}$ holds in neither the current operation mode (indicated by "τ") nor the next operation mode (indicated by "τ+1"). As a result, relation (25) holds.

$$[HoldsAt(II_{i,p},\tau) \wedge HoldsAt(II_{i,p},\tau+1)] \equiv "II_{i,p} = t" \quad (26)$$

Relation (26) shows a *true invariant* that the proof is similar to that of Relation (25).

Table 2. The tabular automaton specification automaton of violations

Current Mode	Event/Invariant								Violated Mode
	Att_1			…	Att_n				
	$II_{1,1}$	…	$II_{i,1}$	…	$II_{1,n}$	…	$II_{k,n}$		
m_1	@T / @F / t / f / x …								Unsafe/ Critical …
…	…								…
m_k	@T / @F / t / f / x …								Unsafe/ Critical …

Table 3. Mapping EC formulae into the tabular

	Compound predicate (ingredients of violation formulae)	Type	Transition ingredient
1	$[\sim HoldsAt(II_{i,p},\tau) \wedge \sim HoldsAt(II_{i,p},\tau+1)]$	invariant	$II_{i,p} = f$
2	$[HoldsAt(II_{i,p},\tau) \wedge HoldsAt(II_{i,p},\tau+1)]$	invariant	$II_{i,p} = t$
3	$Initiates(\alpha, II_{i,p})$ where α is an action	event	initiate mode $II_{i,p}$
4	$\sim HoldsAt(II_{i,p}, \tau) \wedge HoldsAt(II_{i,p}, \tau+1)$	event	$@T(II_{i,p})$
5	$HoldsAt(II_{i,p}, \tau) \wedge \sim HoldsAt(II_{i,p}, \tau+1)$	event	$@F(II_{i,p})$

$$[Initiates(\alpha , II_{i,p})] \equiv \text{Enter mode } II_{i,p} \text{ where } \alpha \text{ is an action} \quad (27)$$

According to Table 1, predicate "Initiates(α, $II_{i,p}$)" asserts, "Event (Action) α initiates fluent $II_{i,p}$. Since each system operation mode is an instance of the system action, we can think of the system action as an attribute, say Att_p, where "$II_{i,p}$" indicates the i^{th} operation mode of the system. This means that "Initiates(α, $II_{i,p}$)" indicates initiating a system operation mode. In the diabetes system, for instance, if Att_p indicates the system action, "Initiates(Compute, $II_{3,p}$)" will indicate that the specific action "Compute" initiates the third operation mode of the system action, say the "Computing" operation mode.

$$[\sim HoldsAt(II_{i,p}, \tau) \wedge HoldsAt(II_{i,p}, \tau+1)] \equiv "@T(II_{i,p})" \quad (28)$$

On the one hand, the right hand side of Relation (28) states, "$II_{i,p}$ is false in the normal mode but it changes to true when the system enters its next mode i.e., unsafe/critical mode". This means, "$II_{i,p}$ is false in the normal mode but it is true in the unsafe/critical mode". On the other hand, based on Relation (18), the left hand side of Relation (28) shows an event. According to Section 4.b, this means, "$II_{i,p}$ does not hold at the τ^{th} occurrence of an attribute but it will hold at the $(\tau+1)^{th}$". In addition, as stated above, the start of every operation mode implies a new occurrence of an attribute; so, we can consider "$\tau+1$" as a new occurrence of an attribute when "τ" indicates its current occurrence. Accordingly, the left hand side of Relation (28) means that $II_{i,p}$ does not hold in the current mode (indicated by "τ"), but it holds in the next mode (indicated by "$\tau+1$"). As a result, relation (28) holds.

$$[HoldsAt(II_{i,p}, \tau) \wedge \sim HoldsAt(II_{i,p}, \tau+1)] \equiv "@F(II_{i,p})" \quad (29)$$

The proof of Relation (29) is similar to that of Relation (28).

To generate elements of the automaton (the last column of Table 3) from the compound predicates (the second column of Table 3), we present an algorithm in Fig. 5 where function "GenerateElement( )" is called for each compound predicate of an EC formulae. The function gets a compound predicate as an input and generates an element of the automaton as an output, i.e. "@T/@F", "t/f", and "Enter mode $II_{i,p}$". The automaton, showing the state-based specification of user's requirements violation, is used to design the run-time monitor.

```
element GenerateElement(cmp: compound)
    e1, e2, e3, e4, e5: element
    e1:= "II_i,p=t",  e2:= "@F(II_i,p)", e3:= "II_i,p=f",  e4:= "@T(II_i,p)", e5: "Enter mode II_i,p" {
  p1:= GetPredicate (cmp);      // take the first predicate of the compound predicate
  p2:= GetPredicate (cmp);      // take the next predicate of the compound predicate
     switch (p1) {
         case "HoldsAt(II_i,p, τ)":
               if (p2 ="HoldsAt(II_i,p, τ+1)")  then return e1;
               if  (p2 = "~HoldsAt(II_i,p, τ+1)") then return e2;
         case "~HoldsAt(II_i,p, τ)":
               if (p2 ="~HoldsAt(II_i,p, τ+1)")  then return e3;
               if (p2 ="HoldsAt(II_i,p, τ+1)")  then return e4;
         case "initiates((Event_i,p)_τ ,II_i,p)": then return e5;
                 }                  // end of switch
}
```

Fig. 5. The algorithm to generate the tabular automaton elements

6. DESIGNING MONITOR (STEP 4)

To design the monitor, we deal with the design of a tabular automaton using *State Design Pattern*. This pattern has already been used for implementing automata [27]. The pattern is suitable for objects whose

behaviors change at run-time [14, 28] i.e., the object behavior depends on run-time events. In addition, the pattern indicates run-time polymorphism i.e., an object behavior is a function of its state. Therefore, we can use the pattern as a method to design the tabular automaton automatically and monitor run-time behavior of software, in accordance with high-level events.

The pattern consists of an abstract class and some concrete classes where the former takes the role of an interface and the latter takes the role of a mode. The abstract class includes the signature of methods as well as a method to control switching between concrete classes where concrete classes include the implementation of signatures. We include a signature for each "@T/@F" of the automaton in the abstract class and implement its signatures in the concrete classes.

As Table 2 shows, each normal mode may have more than one next state (unsafe/critical) under distinct events. Therefore, a concrete class has as many methods as the distinct events that happen in one mode. Passing from an operation mode of automaton to another one is indicated by switching from a concrete class to another one via the interface.

The abstract and concrete classes constitute the monitor program. When an environment event happens, the monitor takes the normal (current) state of target software and the event and switches to the corresponding class using the handler method in the abstract class. In the concrete class, the corresponding method of the event verifies the current state of the target software against the corresponding violations. Then, the monitor issues a warning message of violation if it finds any.

For example, to design a monitor based on tabular automaton shown by Table 4, we consider an abstract class and three concrete classes S_1, S_2, and S_3 where S_1 consists of two methods (a method for each event) and S_2 and S_3 consist of one method (Fig 6.). When, for instance, an environment event happens and the target software enters the state S_1, the monitor takes the state and switches to class S_1 via the state handle method in the abstract class. If the current event is "@T(fluent1)" or "@F(fluent2)", the method "fluent1()" or "fluent2()" in class S1 takes control and verifies invariant "fluent2"/"fluent1" against "f". Then, the monitor will issue a warning message of violation if it finds a matching case.

7. CASE STUDY: CIIP SYSTEM

In this section, to show the effectiveness of our approach, we first introduce the CIIP safety-critical system [15] and then apply the steps of our approach to design a monitor for the system. The system is used for diabetics suffering from "Type 1 Diabetes" called youthful sugar and emerges in all ages, even though children, young people, and ages of before thirty are usually afflicted with the disease [15]. The CIIP system is intended to be worn continuously by a diabetic and the system administers regular doses of insulin based on regular sampling of the wearer's blood-sugar level. Inspired by [15], we have shown the software faults of the CIIP system consisting of computational, algorithmic, and timing in Fig. A.1 (leaf nodes) in the appendix. In the following subsections, we illustrate the steps of our approach to synthesize a monitor for the CIIP system.

a) Specifying domain-based safety requirements

When a physician uses the CIIP system to control a diabetic's blood sugar, he expects, (1) "Blood sugar never falls below 5", (2) "If blood sugar rises above 5, the system delivers insulin", and (3) "The delivered insulin dose should be adequate". According to the physician, blood sugar value varies from 1 to 20. According to the: (1) first requirement, a sugar value below 5 indicates a serious shortfall in blood sugar, (2) second requirement, a sugar value more than 10 indicates excess sugar to be controlled, and (3) third requirement, the delivered insulin dose by the system must be a value not over 6. Among the three mentioned requirements, the violation of the first is *critical* because it leads to a violent affliction so that the system is not able to cope with it and the violations of the second and the third ones are *unsafe* but

remediable. There are two other requirements that are not main requirements of the physician, but are imposed by the physical *constraints* of the system. The first constraint is that, on the one hand, it is not feasible to sample blood sugar constantly and on the other hand, the sampling should not be excessively delayed. Therefore, the physician suggests 10 minute periods to sample the diabetic's blood sugar and expects that it should be satisfied by the system. The second constraint imposed by the system is that the reservoir system may have insufficient insulin to deliver; so, existing insulin should be checked at the time of insulin delivery. The violation of both requirements is unsafe because it is reparable.

Table 4. A sample automaton specification of violations

Current mode	Event/ Invariant			Violated mode
S1	@T	f	x	Unsafe
	f	@F	x	Critical
S2	x	@T	x	Unsafe
S3	@T	t	x	Unsafe

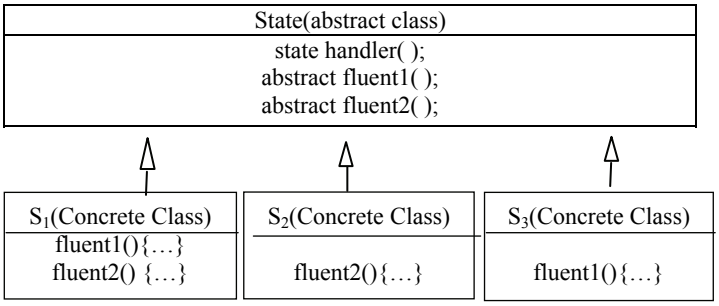

Fig. 6. The class diagram for Table 4

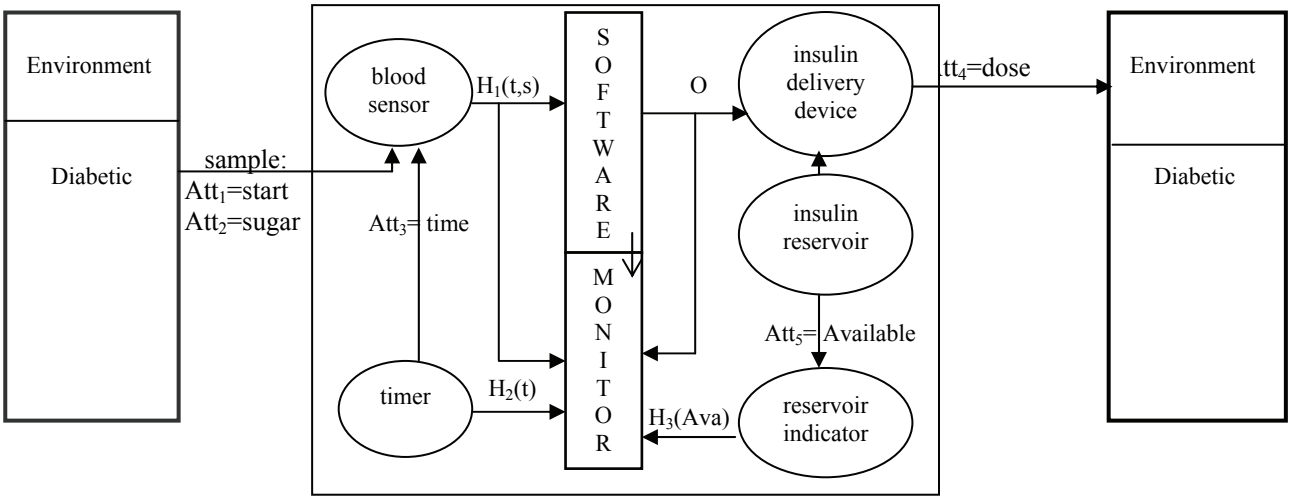

Fig. 7. The CIIP system model

- Modeling system interactions

If we consider the above five requirements, we can designate five attributes: "Start of blood sampling" (Boolean), "Sugar value" (sub-range), "The start time of sampling" (time), "Delivered insulin dose" (integer), and "Available insulin" (sub-range), indicated by Att_1 to Att_5, respectively.

Based on the designated attributes, we present a model for the system and its environment (Fig. 7). The system environment is a diabetic and the system components are input, software, and output devices. The timer device sends an interrupt per ten minutes to the input device (blood sensor) to monitor the environment. The insulin delivery unit steered by the software component is an output device to control the diabetic. Therefore, the CIIP system monitors and controls its environment by means of its components. The environment quantity to be monitored is a *sample* of the diabetic's blood. The system controls its environment through *insulin*, whose dose is determined by the software component and delivered by the output device. The sample quantity has two attributes: *start* of sampling (Att_1), and blood *sugar* (Att_2). Furthermore, insulin quantity has *dose* attribute (Att_4). In addition, the system has two local attributes: *time* (Att_3) and *available* insulin volume (Att_5). While the system acquires Att_1 and Att_2 values from the environment, the environment acquires the Att_4 value from the system. The system mission (i.e., main task) is a computation of correct dose value in a timely manner.

-Data abstraction

The blood sensor converts Att_1 and Att_2 to physical value $H_1(t,s)$ and the reservoir sensor converts Att_5 to physical value $H_3(Ava)$, and they then pass the values to the system software component. The software component computes the output value, which is used by the insulin delivery device to determine the value of Att_4. The *monitor* program receives four values H_1, H_2, H_3, and O from the system devices and the current system state from the software component to analyze them in accordance with safety requirements.

Based on the problem domain of the CIIP system, we first partition the values of each attribute into some intervals (Column "Intervals" of Table 5) and determine their *indicators* (Column "Indicators" of Table 5). Consider Att_2, for instance; according to the problem domain of the CIIP system, Att_2=[1..20] (the second row of Table 5) indicates the set of values of blood sugar where values 5 and 11 denote its boundary values. Accordingly, Att_2 is partitioned into three intervals $I_{1,2}$, $I_{2,2}$ and $I_{3,2}$ and logical variables *low*, *normal,* and *high*, indicated by $II_{1,2}$, $II_{2,2}$, and $II_{3,2}$, are considered as their indicators, respectively (the second row of Table 5). Therefore, low, normal, and high are the abstract values that indicate concrete values 1 to 4, 5 to 10, and 11 to 20, respectively.

Similarly, according to the problem domain, the attribute Att_5=[0..100] (the fifth row of Table 5) indicates available insulin in the system reservoir. The available insulin should not be less than the computed dose. Therefore, the computed dose is a boundary value for Att_5. Based on this boundary value, we partition Att_5 into intervals $I_{1,5}$=[0..dose) and $I_{2,5}$=[dose..100].

$$Att_2 = \{1 , \ldots , 5 , \ldots , 10 , \ldots , 20\} , I_{1,2} = [1 , \ldots 5) , I_{2,2} = [5 , \ldots , 11) , I_{3,2} = [11 , \ldots , 20]$$

-Specifying events, invariants, and safety requirements

Considering Relations (7) and (8) and Relations (15) and (16), we determine events and invariants that violate the user's safety requirements(Column "Violations" of Table 5). Consider abstraction on the "blood sugar" attribute, indicated by Att_2 and its indicators in the second row of Table 5, for instance. Connected with Att_2, the requirement "blood sugar must be normal" is violated if blood sugar falls below normal, indicated by $(Event_{1,2})_\tau=[\sim(II_{1,2})_\tau \wedge (II_{1,2})_{\tau+1}]$ (violation $V_{1,2}$ in Table 5). In addition, connected with Att_2, the invariant $(In_{2,2})_\tau=[(II_{2,2})_\tau \wedge (II_{2,2})_{\tau+1}]$ violates the requirement if $(Event_{2,4})_\tau=[\sim(II_{2,4})_\tau \wedge (II_{2,4})_{\tau+1}]$ happens (violation $V_{2,4}$ in Table 5) and the invariant $(In_{3,2})_\tau=[(II_{3,2})_\tau \wedge (II_{3,2})_{\tau+1}]$ violates the requirement if event $(Event_{1,4})_\tau=[\sim(II_{1,4})_\tau \wedge (II_{1,4})_{\tau+1}]$ happens (violation $V_{3,4}$ in Table 5). Violation $V_{2,4}$ states that the system was going to deliver insulin while the blood sugar was normal and violation $V_{3,4}$ states that the system was not going to deliver sufficient insulin while the blood sugar was high.

According to Section 3.d, a safety requirement, indicated by $S_{i,p}$, is represented as $\sim(Event_{i,p})_\tau$ if $(Event_{i,p})_\tau$ is a violation; so, connected with $V_{1,2}$, $S_{1,2}=\sim(Event_{1,2})_\tau$ is a safety requirement. Also, according to Section 3.d, $(Event_{i,p})_\tau \wedge \sim(In_{j,q})_\tau$ is a safety requirement if $[(Event_{i,p})_\tau \wedge (In_{j,q})_\tau]$ is a violation; so, connected with $V_{2,4}$ and $V_{3,4}$, $S_{2,4}=(Event_{2,4})_\tau \wedge \sim(In_{2,2})_\tau$ and $S_{1,4}=(Event_{1,4})_\tau \wedge \sim(In_{3,2})_\tau$ are two safety requirements, respectively. $S_{1,2}$ assures that blood sugar has not fallen below normal, $S_{2,4}$ assures that the system has not delivered insulin when the blood sugar has been normal, and S_{14} assures that the system has delivered sufficient insulin when the blood sugar has been high.

Similarly, we determine events and invariants for other sets and based on these, we specify the violations of the user requirements of the CIIP system (the last column of Table 5). In column "Formulae" of Table 5, the first part of the formulae indicates an event and the second one (if exists) indicates an invariant. According to the problem domain, among the seven violations that we have described in the last column of Table 5, $V_{1,2}$, $V_{1,4}$, and $V_{2,4}$ are *critical* because they cause blood sugar to fall below 5, leading to irremediable problems, and other violations are *unsafe* because they lead to reparable problems. Negation of the violations constitutes safety requirements for the CIIP system.

Table 5. Domain-based specification of user requirements violation in the CIIP system

p	Att_p	Type	Intervals	Indicators	Violations		
					Name	Formulae	Description
1	start	Bool	Att_1 =[false, true] $I_{1,1}$= [false], $I_{2,1}$ = [true]	$II_{1,1}$ = idle, $II_{2,1}$ = start	$V_{1,1}$	$[\sim(II_{2,1})_\tau \wedge (II_{2,1})_{\tau+1}] \wedge [(\sim II_{2,3})_\tau \wedge (\sim II_{2,3})_{\tau+1}]$	*Sensor is early in sampling:* unsafe
2	sugar	{1..20}	Att_2 =[1..20] $I_{1,2}$=[1..5), $I_{2,2}$=[5..11), $I_{3,2}$ = [11..20]	$II_{1,2}$=low, $II_{2,2}$=normal, $II_{3,2}$=high	$V_{1,2}$	$[\sim(II_{1,2})_\tau \wedge (II_{1,2})_{\tau+1}]$	*Diabetic's sugar has fallen*: critical
3	time	Integer	Att_3 = [1..) $I_{1,3}$= [1..10), $I_{2,3}$= [10] , $I_{3,3}$= [11 ..)	$II_{1,3}$=premature, $II_{2,3}$= timely, $II_{3,3}$ = belated	$V_{1,3}$	$[(II_{2,3})_\tau \wedge (\sim II_{2,3})_{\tau+1}] \wedge [(\sim II_{2,1})_\tau \wedge (\sim II_{2,1})_{\tau+1}]$	*Sensor is late in sampling*: unsafe
4	dose	Integer	Att_4 =[0..) $I_{1,4}$=[0], $I_{2,4}$=[1..6), $I_{3,4}$ =[6 , . . .)	$II_{1,4}$=zero, $II_{2,4}$=ordinary, $II_{3,4}$=over	$V_{1,4}$	$[\sim(II_{3,4})_\tau \wedge (II_{3,4})_{\tau+1}]$	*Pump is going to deliver overdose insulin*: critical
					$V_{2,4}$	$[\sim(II_{2,4})_\tau \wedge (II_{2,4})_{\tau+1}] \wedge [(II_{2,2})_\tau \wedge (II_{2,2})_{\tau+1}]$	*Pump is going to deliver unnecessary insulin*: critical
					$V_{3,4}$	$[\sim(II_{1,4})_\tau \wedge (II_{1,4})_{\tau+1}] \wedge [(II_{3,2})_\tau \wedge (II_{3,2})_{\tau+1}]$	*pump delivers short of insulin:* unsafe
5	volume	{1..100}	Att_5=[0..100] $I_{1,5}$=[0..dose),$I_{2,5}$=[dose..100]	$II_{1,5}$=insufficient $II_{2,5}$=sufficient	$V_{1,5}$	$[\sim(II_{2,4})_\tau \wedge (II_{2,4})_{\tau+1}] \wedge [(\sim II_{2,5})_\tau \wedge (\sim II_{2,5})_{\tau+1}]$	*Reservoir has insufficient insulin*: unsafe

b) Specifying event-based safety requirements

As stated in Section 4, in order to present the event-based specification of safety requirements, we should determine *fluents* in the CIIP system. According to Section 4.a, we show each interval indicator in the CIIP system as a fluent. Table 6 shows fluents for those indicators of the CIIP system which play a role in the user requirements violation. These appear in column "Formulae" of Table 5.

Table 6. Description of *used* fluents in the CIIP System

	Fluent	II	Description: Value
1	start	$II_{2,1}$	*start of sampling: true if started*
2	low	$II_{1,2}$	*blood sugar fallen: true if sugar <5*
3	normal	$II_{2,2}$	*blood sugar normalized: true if 4< sugar<11*
4	high	$II_{3,2}$	*blood sugar rose: true if sugar>10*
5	timely	$II_{2,3}$	*time of sampling: true if time = 10*
6	zero	$II_{1,4}$	*no dose is deliverable: true if dose = 0*
7	ordinary	$II_{2,4}$	*dose delivery is normal: true if 0<dose<6*
8	over	$II_{3,4}$	*dose delivery is over: true if dose >5*
9	sufficient	$II_{2,5}$	*available insulin in reservoir: true if dose≤available*

Table 7. The event-based specification of violations in the CIIP system

Name	Description	Violation Formulae
$V_{1,1}$	early sampling	*Initiates(@T(start), Unsafe)← [HoldsAt(~start, τ) ∧ HoldsAt(start, τ+1)]∧ [HoldsAt(~timely, τ) ∧ HoldsAt(~timely, τ+1)]*
$V_{1,2}$	blood sugar falling	*Initiates(@T(low), critical)← [HoldsAt(~low, τ) ∧ HoldsAt(low, τ+1)]*
$V_{1,3}$	late sampling	*Initiates(@T(timely), Unsafe)← [HoldsAt(timely, τ) ∧ HoldsAt(~timely, τ+1)]∧ [HoldsAt(~start, τ) ∧ HoldsAt(~start, τ+1)]*
$V_{1,4}$	overdose delivery	*Initiates(@T(over),Critical) ← [~HoldsAt(over, τ) ∧ HoldsAt(over, τ+1)]*

Table 7. Continued.

$V_{2,4}$	needless dose delivery	*Initiates(@F(ordinary),Critical)← [HoldsAt(~ordinary, τ) ∧ HoldsAt(ordinary, τ+1)] ∧ [HoldsAt(normal, τ)∧ HoldsAt(normal, τ+1]*
$V_{3,4}$	short of dose delivery	*Initiates(@T(zero),Unsafe) ← [HoldsAt(~zero, τ) ∧ HoldsAt(zero, τ+1)] ∧ [HoldsAt(high, τ) ∧ HoldsAt(high, τ+1)]*
$V_{1,5}$	Insufficient dose	*Initiates(@T(ordinary),Unsafe)← [~HoldsAt(~ordinary, τ) ∧ HoldsAt(ordinary, τ+1)] ∧ [HoldsAt(~sufficient, τ)∧ HoldsAt(~sufficient, τ+1)]*

Table 8. The tabular automaton specification of violations in the CIIP system

Name	Current mode	Violation (Event / Invariants)									Violated Mode
		start	timely	low	normal	high	zero	ordinary	over	sufficient	
$V_{1,1}$	Inactive	@T	f	-	-	-	-	-	-	-	Unsafe
$V_{1,3}$		f	@F	-	-	-	-	-	-	-	Unsafe
$V_{1,2}$	Sampling	-	-	@T	-	-	-	-	-	-	Critical
$V_{1,4}$	Computing	-	-	-	-	-	-	-	@T	-	Critical
$V_{2,4}$		-	-	-	t	-	-	@T	-	-	Critical
$V_{3,4}$		-	-	-	-	t	@T	-	-	-	Unsafe
$V_{1,5}$		-	-	-	-	-	-	@T	-	f	Unsafe

- Specifying events, invariants, and safety requirements

As we stated in Sections 4.b and 4.c, we specify events and invariants using predicates, "HoldsAt" and "Initiates" in EC. Among the events and invariants, those that violate user requirements of the CIIP system are our concern, as stated in the last column of Table 5. For each violation in Table 5, a row has been intended in Table 7 where its premise part (the right hand side) shows the conditions of the violation and its conclusion part (the left hand side) shows the cause and significance of the violation with safety or critical. "*@T*(fluent)" or "@F(fluent)" denotes that the truth-value of a fluent has changed from false/true to true/false. The important point to be considered is that each formula in Table 7 ties events (the right hand side) to states (the left hand side); this is a bridge that helps us to move to state-based specification.

c) Specifying state-based safety requirements

The CIIP system has four operation modes: (1) "Inactive", (2) "Sampling", (3) "Computing" and (4) "Delivering" where "Inactive Mode" is an initial mode. As we stated in Section 5, we use the algorithm in Fig. 5 to generate the elements of tabular automaton of the CIIP system, i.e., "@T", "@F", "t", and "f" (Table 8). Each row of Table 8 corresponds to a row in Table 7. Consider violation $V_{1,1}$ in Table 8, for instance. "@T" under column "start" and "f" under column "timely" have been generated from the first and the second compound predicates of $V_{1,1}$ violation formulae in Table 7, respectively. Note that there is no violation of user requirements in the "Delivering" mode and so there is no entry for this mode in Table 8.

Each row in Table 8 shows a state-based specification of the violation of a user requirement of the CIIP system where column "Violation" indicates the transition makes a violation. As stated in Section 5, we decide on the system operation modes thinking of the attributes. According to Table 5, there are five attributes that operations should be determined for to specify their values. We designate operation modes: "Idle" to specify the values of Att_1 and Att_3, "Sampling" to specify the value of Att_2, and "Computing" to specify the values of Att_4 and Att_5. The specification, "Inactive @T f Unsafe" in the first row of Table 8, for instance, states, "When the CIIP system is performing the wait operation (the *Inactive* mode), if event "*@T(start)*" happens and invariant "*timely*" does not hold, a violation will happen". In other words, this

specification states, "If the blood sampling event starts early when the system is in the Inactive mode, the system violates a user requirement".

d) Designing CIIP monitor

Based on Table 8, we have depicted a visual automaton of requirements violation in the CIIP system in Fig. 8 where each state indicates a mode of the automaton operation. The solid and dotted ellipses indicate normal and unsafe/critical modes, respectively. In keeping with the view of normal modes, the monitor program for the CIIP system will consist of three concrete classes, "Inactive", "Sampling" and "Computing", consisting of two, one, and four methods respectively. When the system enters the "Inactive" mode, the monitor creates an object of the "Inactive" class. When event @T(start) happens, the related method is invoked to check the value of $II_{2,3}$ (i.e. the fluent "timely"). If it is false, the monitor will announce a violation.

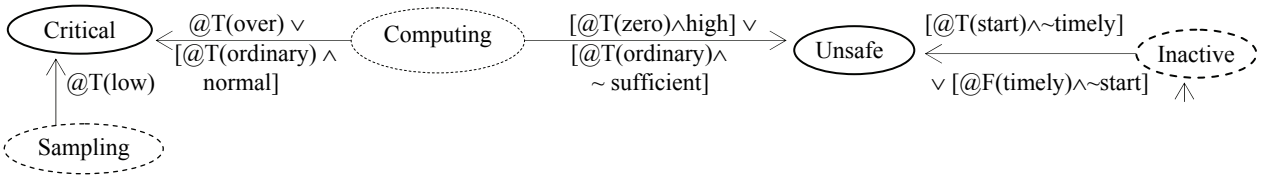

Fig. 8. A visual automaton of violations in the CIIP system

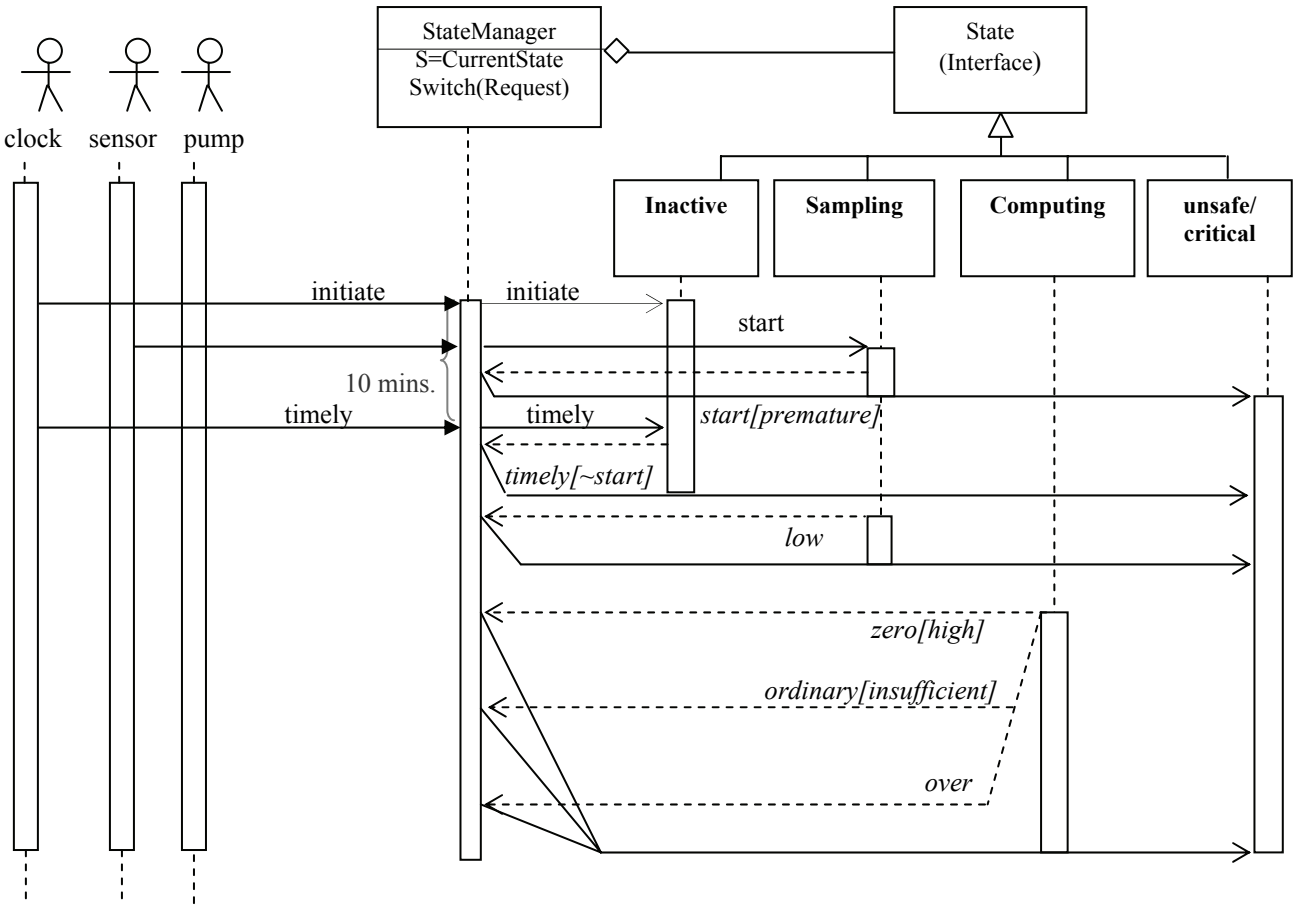

Fig. 9. The event tracing sequence diagram of violations in the CIIP system

Figure 9 shows the sequence diagram of the monitor behavior. When an actor, i.e., "Clock", "Sensor", or "Pump" generates an event, the monitor, *StateManager*, creates an object of a concrete class and

delegates the control to the related method. The method checks related invariants and returns a message to the *StateManager* class. If the message denotes an incorrect system behavior, the *StateManager* creates the object of an Unsafe/Critical class and delegates the announcement to the *announce* method of the class. The solid lines indicate sending messages, and dash lines indicate returning ones.

e) Evaluation of effectiveness of the approach

In the previous subsections, we employed three steps in processing a diabetic's safety requirements in order to derive the monitor. In this section, we evaluate the effectiveness of these steps. (1) the *effectiveness of requirements documentation*. Related to safety-critical systems, the system's expert in the application domain usually reviews the requirement documents to ensure that they are consistent and correct. Such reviews will be *effective* if the documents are clear and understandable to the expert. To meet this feature, in Section 7.a we expressed domain-based safety requirements in the physician's language and dealt with the abstraction of diabetic quantities that were to be monitored and controlled. (2) the *effectiveness of requirements specification*. In the case of safety-critical systems, specification of the requirements should be verifiable formally so that one can show its correctness and consistency. To provide such a specification, we exploited a first order logic called Event Calculus and mapped the documented requirements of the diabetic to EC predicates in Table 7. (3) *effectiveness of requirements design*. By mapping the event-based specification of a diabetic's safety requirements to an automaton, we provided an abstract model that provided an effective design to implement the monitor. To show the effectiveness of the design, it was employed to implement the monitor of a diabetic's safety requirements in Section 7.d.

8. CONCLUSIONS AND RELATED WORK

In this paper, an approach to synthesize the run-time monitor systematically in order to verify the run-time behavior of safety-critical systems against safety requirements was proposed. Our approach commenced with the problem domain at user level, expressing user requirements in his/her vocabulary, and finally culminated in synthesizing the run-time monitor in a systematic manner. Table 9 shows an overview of our approach where its features are discussed in the following:

(1) Our approach presented a constructive method to map abstract and high-level safety requirements specification to concrete and low-level software activities where event-based and state-based formal methods were used. Other approaches map high-level specifications to low-level ones by a direct and specific way. The Java-Mac method specified high-level requirements by the MEDL specific language and mapped them to the PEDL low-level specific language [29]. Jahanian and Mok specified high-level requirements by the Modechart visual and specific language and mapped them to a constraint graph [30] where the graph was used by a monitor in order to verify software run-time behavior. Chen and Rosu presented a paradigm called monitoring oriented programming (MoP) to combine a high-level formal specification with low-level implementations [31]. Although the MoP approach allows one to select a high-level formal specification, it annotates the specifications at user-selected places in java programs and hence it does not separate the specification from the program implementation to fully satisfy abstraction.

(2) Deriving the monitor from requirements is a challenge faced by synthesizers of the monitor. To face the challenge: (1) The Eagle method used the Eagle logic to specify requirements and synthesized a rule based monitor [32]. It extended Mu Calculus to support past and future time linear logic and real-time logic and (2) used real-time logic (RTL) for the requirements specification of a real-time system [33, 30]. Nevertheless, these approaches have the limitation of unfamiliarity and strangeness of system users with formal specification methods. As a result, the users cannot use the methods to state their requirements. Therefore, an appropriate specification of requirements is noteworthy and significant. This is why we specified the safety requirements of safety-critical systems at user-level and then formalized them. To

overcome the limitation, we proposed the formal specification of safety requirements through the domain-based specification, presented by the language of system experts. By this means, we isolated the system from its environment and specified the environment from the user's point of view.

Connected with deriving the monitor, the software testing approach also has a method called *oracle-based* testing where an oracle program is derived from the requirements specification and is attached to the target software under test. The oracle weighs the target program outputs against the expected outputs [34]. Leucker and Schallhart in the Section "Runtime Verification versus Testing", stated that in essence, runtime verification can be understood as oracle-based testing, but it does not involve generating a suitable set of input sequences to exhaustive testing of the target software [17]. They conclude that run-time verification can be considered as a form of passive testing.

A combination of software testing techniques with run-time verification is an interesting approach that has drawn attention. Bouquet et al used software testing techniques to derive test cases from formal specifications stated in Java Modeling Language in order to verify run-time safety properties [35]. Also, Li and Dang presented algorithms to combine automata with black-box testing in order to verify safety properties [36]. Arto et al systematically generated test cases based on the program's input domain and then verified execution traces against temporal logic specifications [37]. There are two main approaches to generate test inputs automatically, (1) from a model of the system, which is a static approach and (2) by executing the program repeatedly, while employing criteria to rank the quality of the test produced, which is a dynamic approach.

(3) Our approach presented an external view, event-based specification, and an internal view, state-based one, of safety requirements violations; accordingly, it can be used for both event-based and state-based systems. However, some approaches provide only state-based specifications and implement them in an ad-hoc manner [38]. "Eagle" exploiting a state-based approach [39] was extended by the HAWK logic [39] in order to support event-based specifications. The HAWK language, which is used to specify both high-level specifications and low-level run-time states, has included low-level programming definitions to verify java programs. As a result, it requires a lower level of abstraction and clarity for event specification in comparison with our approach. However, the domain-based specification of our approach allows a clear and transparent specification to be presented according to the system users' language [28, 33, 40] exploiting the event-based approach for real-time systems, and using real-time logic (RTL) for the high-level specification of the system requirements; so, they were capable of specifying real-time requirements, particularly the ordering of events. However, by means of RTL, one can only express the event-based specification, hence it is not adequate for state-based requirements.

Table 9. Summary of the proposed approach

	Step name	Used method	Task	Style	Feature
1	Domain-based specification	data abstraction	designating *attributes*	manual	- definition of *events* and *invariants*
2	Event-based specification	Event Calculus	determining *fluents* and *predicates*	automatic	-formalization of *interactions* between system and environment -formalization of *high-level* requirements *violations*
3	State-based specification	tabular automaton	determining *states* and *transitions*	automatic	-formalization of *low-level* requirements *violations*
4	Monitor design	State Design Pattern	determining *classes* and *methods*	pattern-based	- facilitation of implementing *dynamic behavior*

(4) Using the State Design Pattern, our approach dealt with mapping the abstract system operation modes to concrete ones and thus addressed the automatic design of the monitor. This is fulfilled by converting the tabular automaton to its concrete counterpart automatically.

(5) Our approach is a constructive approach that separates a safety-critical system from its environment; accordingly, this helps us to extend it to support safety requirements in *agent-based* systems. To this, we suggest: (a) specifying safety requirements by means of KAOS [41] instead of an event-based one in the second step of our approach and (b) using the method has been presented in [42] in order to map goal-oriented specifications into the tabular automaton in the third step of our approach. The first and the fourth steps of our approach have remained unchanged. Using goal-oriented specification of safety requirements, one can specify who is responsible for the safety requirements violations. In this specification, goals are separated from *features*. Goals are prescriptive statements of the required and expected software behavior, while features are environment specifications such as physical rules, constraints and organizational procedures and policies.

In order to organize the above wide-ranging discussion, we categorize it based on three keystones: (1) *taking advantage of two kinds of specifications* to demonstrate two views of behavior of safety critical systems, one to demonstrate interactions of the system to its environment in terms of events, and the other to demonstrate its internal behavior in terms of states. Tying these two specifications, we met the challenge of mapping high-level specifications into low-level ones in order to derive the monitor from high-level safety requirements specification. (2) *taking advantage of three hierarchical levels of specifications*, user level, specification level, and implementation level to provide a constructive method to synthesize the monitor. The *abstraction* of the specification of the safety requirements and the *extension* of the approach usage are two corollaries of the hierarchical level of specifications, as distinct level specifications together with the hierarchical structure enable us to support *goal-oriented systems*. To this, one can replace the event-based specification by a goal-oriented one and then map it to an automaton [41]. (3) *taking advantage of state design pattern* to facilitate the implementation of the dynamic behavior of objects from abstract operations specified by the tabular automaton.

REFERENCES

1. Knigh, J. C. (2002). Safety critical systems: challenges and directions. *In Proceedings of the 24th International ACM/IEEE Joint Conference on Software Engineering*, (547-550).
2. Nelson, S. (2003). Certification processes for safety-critical and mission-critical aerospace software. *NASA/CR–2003-212806*.
3. Saffarzadeh, M. & Farshad, S. (2005). Repair and maintenance models for highway traffic control devices. *Iranian Journal of Science & Technology, Transaction B: Engineering*, Vol. 29, No. B3, pp. 289-299.
4. O'Connor, B. (2004). NASA software safety guidebook. NASA-GB-8719.13.
5. Leucker, M. & Schallhart, C. (2009). A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, Elsevier, Vol. 78, No. 5, pp. 293-303.
6. Rosu, G. et al. (2009). Runtime verification of C memory safety. *In Proceedings of the 9th International Workshop on Run-Time Verification, RV 2009*, Springer, LNCS 5779, pp. 132-151.
7. Gahl, D. J. et al. (1972). Notes on structured programming. *Academic Press London*.
8. Parnas, D. L. et al. (1990). Evaluation of safety-critical software. *Communications of the ACM*, Vol. 33, No. 6, pp. 636-648.
9. Littlewood, B. & Stringini, L. (1993). Validation of ultrahigh dependability for software-based systems. *Communications of ACM*, Vol. 11, No. 36, pp. 69-80.

10. Buttler, R. W. & Finclli, G. B. (1993). The infeasibility of quantifying reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, Vol. 19, pp. 3-12.
11. Iyer, R. K. & Verlardi, P. (1986). Hardware-related software errors: measurement and analysis. *IEEE Transaction on Software Engineering*, Vol. 11, No. 2, pp. 223-231.
12. Kowalski, R. A. & Sergot, M. J. (1986). A logic based calculus of events. *New Generation Computing*, Vol. 4, pp. 67-95.
13. Sadri, F. & Kowalski, R. (1995). Variants of the event calculus. *In Proceedings of the 12th International Conference on Logic Programming (ICLP)*, MIT Press, pp. 67-82.
14. Gamma, E. et al. (1995). *Design patterns, elements of reusable object-oriented software*. Addison-Wesley.
15. Babamir, S. M. (2010). *A framework for specification and verification of the CIIP medical system behavior*. *Advances in Experimental Medicine and Biology*, Vol. 696: Software Tools and Algorithms for Biological Systems, Springer-Berlin, 550 pp, ISBN: 978-1-4419-7045-9.
16. Rtho, C. et al. (2005). Combining test case generation and runtime verification. *Journal of Theoretical Computer Science,* Elsevier, Vol. 336, Nos. 2-3, pp. 209-234.
17. Leucker, M. & Schallhart, C. (2009). A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, Elsevier, Vol. 78, No. 5, pp. 293-303.
18. Brat, G. et al. (2004). *Experimental evaluation of verification and validation tools on Martian Rover software*. Formal Methods in System Design. Kluwer Academic Publishers, Vol. 25, Nos. 2-3), pp. 167-198.
19. Barringer, H. (2007). From runtime verification to evolvable systems. *In the 7th International Workshop On Run-time Verification, RV 2007*, Lecture Notes in Computer Science, Springer, 4839, pp. 97-110.
20. Zee, K. (2007). Runtime checking for program verification. *In the 7th International Workshop On Run-time Verification, RV 2007*, Lecture Notes in Computer Science, Springer, 4839, pp. 202-213.
21. Delgado, N. et al. (2004). Taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, Vol. 30, No. 22, pp. 859-872.
22. Pnueli, A. (1977). The temporal logic of programs. *In the18th annual IEEE-CS Symposium on Foundations of Computer Science*, pp. 46-57.
23. Parnas, D. L. & Madey, J. (1995). Functional documents for computer systems. *Science of Computer Programming*, Vol. 25, No. 1, pp. 41-61.
24. Lipschutz, S. (1998). *Schaum's outline of theory and problems of set theory and related topics*. McGraw-Hill.
25. Mueller, E. T. (2006). *Commonsense resoning*. Elsevier.
26. Heitmeyer, C. L. et al. (1996). Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, pp. 231–261.
27. Samek, M. (2002). *Practical statecharts in C/C++, quantum programming for embedded systems*. CMP Books.
28. Douglass, B. P. (1999). *Doing hard time, developing real-time systems with UML, objects, framework and patterns*. Addison-Wesley.
29. Kim, M. et al. (2004). Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in Systems Design*, Vol. 24, No. 2, pp. 129-155.
30. Jahanian, F. & Mok, A. K. (1994). Modechart: a specification language for real-time systems. *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, pp. 933-947.
31. Chen, F. & Rosu, G. (2005). Java-MOP: a monitoring oriented programming environment for Java. *In Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 546-550.
32. Barringer, H. et al (2004). Rule-based runtime verification. *In Proceedings of the 5th International Conference on Verification*, *Model Checking and Abstract Interpretation (VMCAI'04),* LNCS2937, Springer, pp. 44-57.
33. Mok, A. K. & Liu, G. (1997). Efficient run-time monitoring of timing constraints. *In IEEE Real-Time Technology and Applications Symposium*, Vol. 252-262.

34. Wang, X., Qi, Z., & Li, S. (2005). An optimized method for automatic test oracle generation from real-time specification. *In Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 440-449.
35. Bouquet, F. (2006). Safety property driven test generation from JML specifications. *In Proceedings of Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006*, Springer, Vol. 4262.
36. Li, C. & Dang, Z. (2006). Decompositional algorithms for safety verification and testing of aspect-oriented systems. *In Proceedings of Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006*, Springer, Vol. 4262.
37. Artho, C. et al. (2003). Experiments with test case generation and runtime analysis. *In the 10th International Workshop on Abstract State Machines*, Springer, LNCS, Vol. 2589/2003, pp. 87-108.
38. Peters, D. K. & Parnas, D.L. (2002). Requirements-based monitors for real-time systems. *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, pp. 146-158.
39. d'Amiron, M. & Havelund, K. (2005). Event-based runtime verification of Java programs. *In Proceedings of the 5th Workshop of ICSE (International Conference on Software Engineering) on Dynamic Analysis*, pp. 1-7.
40. Jalili, S. & MirzaAghaei, M. (2007). RVERL: run-time verification of real-time and reactive programs using event-based real-time logic approach. *In the 5th International Conference on Software Engineering Research, Management & Applications (SERA 2007),* IEEE Computer Society, pp. 550-557.
41. Lamsweerde, A. V. & Letier, E. (2000). Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, Vol. 26, No. 10, pp. 978-1005.
42. Landtsheer, R. D. et al. (2003). Deriving tabular event-based specifications from goal-oriented requirements models. *In Proceedings of RE'03, the 11th IEEE International Conference on Requirements Engineering* (Best Research Paper Award), pp. 104-120.

Appendix

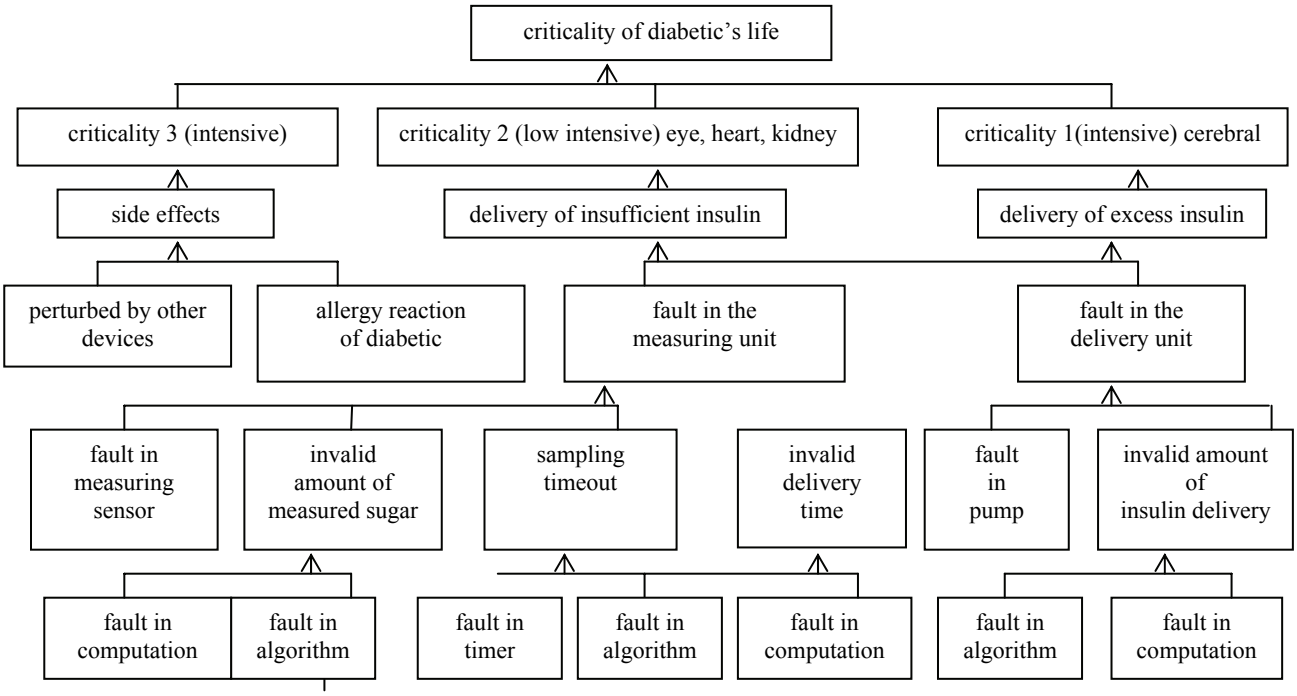

Fig. A.1. The CIIP system fault tree