Model Checking For Fault Explanation

Shengbing Jiang and Thomas E. Fuhrman
GM R&D, MC: 480-106-390
30500 Mound Road
Warren, MI 48090-9055
Email: shengbing.jiang, thomas.e.fuhrman@gm.com

Sumit K. Jha
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
Email: jha+@cs.cmu.edu

***Abstract*— Model checking is very effective at finding out even subtle faults in system designs. A counterexample is usually generated by model checking algorithms when a system does not satisfy the given specification. However, a counterexample is not always helpful in explaining and isolating faults in a system when the counterexample is very long, which is usually the case for large scale systems. As such, there is a pressing need to develop fault explanation and isolation techniques. In this paper, we present a new approach for the fault explanation and isolation in discrete event systems with LTL (linear-time temporal logic) specifications. The notion of *fault seed* is introduced to characterize the cause of a fault. The identification of the fault seed is further reduced to a model checking problem. An algorithm is obtained for the fault seed identification. An example is provided to demonstrate the effectiveness of the approach developed.**

I. Introduction

Model checking ([3]) of a faulty system against a suite of temporal logic specifications produces a counterexample to those specifications which are not satisfied by the system. The counterexample produced by model checking algorithms has traditionally been accepted as the final result of the "bug-finding" mission. However, it is now being realized that the counterexample is not always helpful in explaining and isolating faults in a system when the counterexample is very long, which is usually the case for large scale systems. As such, there is a pressing need to develop fault explanation and isolation techniques.

It has been shown in [1] that correct execution traces can be used to localize the cause of the fault in a counterexample trace. The use of the closest correct trace to a counterexample trace for isolating the fault has also been suggested in [5], [6]. There has also been an attempt ([2]) to generate abstract explanations for the faults by using the predicates involved in the abstraction instead of the actual values of the atomic propositions in each state of the counterexample trace.

In this paper, we present a new approach for the fault explanation and isolation in discrete event systems with LTL (linear-time temporal logic) ([3]) specifications. Generally speaking, the fault explanation and isolation is to identify the cause of a fault. In other words, we have the relation of *cause* $\Rightarrow$ *fault*, and now we have a fault, which is represented by the counterexample, and we want to identify the cause. Then the question comes to the definition of the cause, i.e., what is the cause and how to characterize the cause.

In discrete event systems, a fault is usually caused by the execution of a small set of transitions in a certain order. With the above observation, we use the notion of *fault seed* to characterize the cause of a fault. Intuitively, a fault seed is an ordered set of transitions such that any trace of the system containing these transitions in the set by the given order is a faulty trace, where a trace is called a faulty trace if it violates the given specification. Now the fault explanation becomes the identification of the fault seed. The identification of the fault seed is further reduced to a model checking problem. More precisely, given a candidate for the fault seed, we first encode the candidate by a LTL formula (called candidate formula), and then model check the system for the property of "*candidate formula* $\Rightarrow \neg$*Specification*". A candidate is a real fault seed if and only if the system satisfies the property *candidate formula* $\Rightarrow \neg$*Specification*. To obtain the candidates for the fault seed, we use a brute-force approach based on the facts that the length of a fault seed is usually short and a long fault seed will not provide much help for the explanation of the fault. So we list all possible ordered sets of transitions up to a given length bound as the candidates for the fault seed and then perform the model checking for each candidate until a real fault seed is found.

Note that the problem of fault explanation and isolation studied here is different from the problem of failure diagnosis studied in [8]. In [8], the failure diagnosis problem of discrete event systems with LTL specifications was studied; and the problem was to diagnosis the occurrence of the faults (violations of the LTL specification) while the system is in operation based on limited observations of the system behavior. For the problem of fault explanation and isolation studied here, we already know that there is a fault in the system (represented by the counterexample), but we do not know what and where the actual fault is, and we need to identify what and where the fault is based on the counterexample and the model of the system, and there is no issue of partial observation.

The rest of the paper is organized as follows. In Section 2, we present some notations and preliminaries. In Section 3, we study the problem of fault explanation. We first introduce the notion of fault seed and then reduce the fault seed identification problem to a model checking problem. In Section 4, we provide an illustrative example.

II. Notations and Preliminaries

In this paper, we use LTL temporal logic to express the specifications of discrete event systems. In the following, we give the definition of LTL. For a complete introduction to temporal logic, readers may refer to [4].

Let AP be a finite set of atomic proposition symbols. Using the atomic propositions and boolean connectives such as conjunction, disjunction, and negation, one can construct more expressions describing properties of states. However one is also interested in describing the properties of sequences of states. Such properties are expressed using *temporal operators* of a temporal logic. LTL temporal logic is a specific temporal logic formalism. The following temporal operators are used in LTL for describing the properties along a specific state-trace: X ("next time"), U ("until"), F ("eventually" or "in the future"), G ("globally" or "always"), and B ("before").

LTL formulae are generated by rules P1-P3 given below.

- P1 If $p \in AP$, then p is a LTL formula.
- P2 If f_1 and f_2 are LTL formulae, then so are $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$.
- P3 If f_1 and f_2 are LTL formulae, then so are Xf_1, $f_1 U f_2$, Ff_1, Gf_1, and $f_1 B f_2$.

Next we give the semantics of LTL, which is defined over infinite proposition-traces. Let $\Sigma_{AP} = 2^{AP}$, then Σ^*_{AP} be the set of all finite proposition-traces over AP, and Σ^ω_{AP} be the set of all infinite proposition-traces over AP. For a LTL formula f and $\pi \in \Sigma^\omega_{AP}$, the notation $\pi \models f$ (resp., $\pi \not\models f$) means that f holds (resp., does not hold) along the infinite proposition-trace π. The relation $\models$ is defined inductively as follows, where we assume that f_1 and f_2 are LTL formulae, and for $\pi = (L_0(\pi)L_1(\pi)\cdots) \in \Sigma^\omega_{AP}$, $\pi^i = (L_i(\pi)\cdots)$ for any $i \geq 0$.

1) If $f_1 \in AP$, then $\pi \models f_1 \iff f_1 \in L_0(\pi)$.
2) $\pi \not\models f_1 \iff \neg(\pi \models f_1)$.
3) $\pi \models \neg f_1 \iff \pi \not\models f_1$.
4) $\pi \models f_1 \vee f_2 \iff \pi \models f_1$ or $\pi \models f_2$.
5) $\pi \models f_1 \wedge f_2 \iff \pi \models f_1$ and $\pi \models f_2$.
6) $\pi \models Xf_1 \iff \pi^1 \models f_1$.
7) $\pi \models f_1 U f_2 \iff \exists k \geq 0$, $\pi^k \models f_2$ and $\forall j \in \{0, 1, \cdots, k-1\}$, $\pi^j \models f_1$.
8) $\pi \models Ff_1 \iff \exists k \geq 0$, $\pi^k \models f_1$.
9) $\pi \models Gf_1 \iff \forall k \geq 0$, $\pi^k \models f_1$.
10) $\pi \models f_1 B f_2 \iff \forall k \geq 0$ with $\pi^k \models f_2$, $\exists j \in \{0, 1, \cdots, k-1\}$, $\pi^j \models f_1$.

A discrete event system is a 6-tuple $M_{DES} = (\mathcal{S}, \Sigma, \mathcal{R}, s_0, \mathcal{L}, \mathcal{AP})$, where $\mathcal{S}$ is the state set, Σ is the event set, $R \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ is the transition relation, s_0 is the initial state, $\mathcal{AP}$ is the set of atomic propositions, and $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{AP}}$ is the labeling function that labels each state to a set of atomic propositions held in the state. Let S be a given LTL specification over the atomic proposition set $\mathcal{AP}$.

A *state trace* (either finite or infinite) of the system M_{DES} is a sequence of states $t = (s_0, s_1, s_2, \cdots)$ such that $\forall i = 1, \cdots, \exists \sigma_i$, $(s_{i-1}, \sigma_i, s_i) \in \mathcal{R}$. $(\sigma_1, \sigma_2, \cdots)$ is called the Σ trace (also called event trace) associated with the state trace t; and $(\mathcal{L}(s_0), \mathcal{L}(s_1), \cdots)$ is called the $\mathcal{AP}$ trace associated with t.

The system M_{DES} can be transferred into a standard Kripke structure M ([3]) as follows:

$$M = (X, R_\Sigma, x_0, \mathcal{L}_\Sigma, \mathcal{AP}_\Sigma),$$

where $X = \mathcal{S} \times (\Sigma \cup \{\epsilon\})$ is the state set (ϵ is the null event), $R_\Sigma \subseteq X \times X$ is the transition relation with $R_\Sigma = \{((s, \sigma), (s', \sigma')) | (s, \sigma', s') \in \mathcal{R}\}$, $x_0 = (s_0, \epsilon)$ is the initial state, $\mathcal{AP}_\Sigma = \mathcal{AP} \bigcup \Sigma$ is the new atomic proposition set, $\mathcal{L}_\Sigma : X \to \mathcal{AP}_\Sigma$ is the labeling function such that $\forall (s, \sigma) \in X$, $\mathcal{L}_\Sigma((s, \sigma)) = \mathcal{L}(s) \cup \{\sigma\}$.

For a state trace $t = ((s_0, \epsilon), (s_1, \sigma_1), \cdots)$ in M, its associated Σ event trace is $t^\Sigma = (\epsilon, \sigma_1, \sigma_2, \cdots)$, its associated $\mathcal{AP}$ trace is $t^{AP} = (\mathcal{L}(s_0), \mathcal{L}(s_1), \cdots)$, and its associated $\mathcal{AP}_\Sigma$ trace is $t^{AP_\Sigma} = (\mathcal{L}(s_0), \mathcal{L}(s_1) \cup \{\sigma_1\}, \cdots)$,

It is not difficult to find out that the σ labeled Kripke structure M has the same sets of Σ and $\mathcal{AP}$ traces as the original discrete event system M_{DES}. Thus, from now on, we only consider the σ labeled Kripke structure system M.

Let T_M denote the set of all infinite length state traces of M. $\forall t \in T_M$, t is said to be a *faulty trace* with respect to the LTL specification S if $t^{AP} \not\models S$, which is also denoted as $t \not\models S$. (Note that in this paper we require that the system M is non-terminating, i.e., at every state of the system there exists at least one out-going transition. This is because the semantics of LTL is defined over the traces of infinite length.)

A finite trace $ce = (x_0, x_1, x_2, \cdots, x_m)$ of M is called a *counterexample* with respect to the LTL specification S if the trace is a witness of the violation of the specification S. More precisely, if S is a safety specification then ce is a counterexample if and only if any arbitrary infinite extension of ce^{AP} violates the specification S, i.e., $\forall t_0 \in (2^{AP})^\omega$, $ce^{AP} \cdot t_0 \not\models S$; if S specifies some liveness property then ce is a counterexample for S if and only if there is a x_i ($0 \leq i \leq m$) in ce such that $x_i = x_m$ and the infinite extension of ce along the cycle $(x_i, x_{i+1}, \cdots, x_{m-1}, x_m = x_i)$ violates the specification S, i.e., $ce(x_{i+1}, \cdots, x_m)^\omega \not\models S$. In order to distinguish the above two cases, if S is a safety specification then the counterexample ce is written as $ce = ce_{safe} = (x_0, \cdots, x_m)$; otherwise ce is written as $ce = ce_{live} = ((x_0, x_1, \cdots, x_i)), (x_{i+1}, \cdots, x_m))$ with $x_m = x_i$.

III. Fault Explanation

As we stated above, when a system fails the model checking, a counterexample will be generated by the model checker. To explain the violation of the specification based on the counterexample generated, we want to identify those transitions (represented by the transition labels) in the counterexample such that any trace including the ordered executions of those transitions is faulty. Those transitions are captured by the notion of *fault seed* defined below.

Definition 1: Given a system M, a LTL specification S, a counterexample ce, and an event trace sd ($sd = (e_1, \cdots, e_k)$ if $ce = ce_{safe} = (x_0, \cdots, x_m)$; $sd = ((e_1, \cdots, e_q), (e_{q+1}, \cdots, e_k))$ if $ce = ce_{live} =$

$((x_0, \cdots, x_i), (x_{i+1}, \cdots, x_m)))$, sd is called a *fault seed* for the counterexample ce if the following holds:

1) sd is orderly executed by ce.
   The orderly execution of sd by ce is defined as follows. For the case of $ce = ce_{safe}$, let $(\sigma_1, \cdots, \sigma_m)$ be the event trace associated with ce, then $sd = (e_1, \cdots, e_k)$ is called *orderly executed* by ce if there exists a sequence of integers $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ such that $\forall j = 1, \cdots, k$, $\sigma_{i_j} = e_j$ and $\forall j = 1, \cdots, k-1$:

   $$\{\sigma_{i_j+1}, \sigma_{i_j+2}, \cdots, \sigma_{i_{j+1}-1}\} \cap \{e_j, e_{j+1}, \cdots, e_k\} = \emptyset$$

   For the case of $ce = ce_{live}$, $sd = ((e_1, \cdots, e_q), (e_{q+1}, \cdots, e_k))$ is called *orderly executed* by ce if $(e_1, \cdots, e_q)$ is orderly executed by $(x_0, \cdots, x_i)$ and $(e_{q+1}, \cdots, e_k)$ is orderly executed by $(x_i, x_{i+1}, \cdots, x_m)$.
2) For any infinite trace t in the system, if sd is orderly executed by t as by ce, then t is faulty.
   The orderly execution of sd by an infinite trace t as by ce is defined as follows. For the case of $ce = ce_{safe}$, sd is said to be orderly executed by t as by ce if there exists a finite prefix of t such that sd is orderly executed by the finite prefix as defined above for the case of $ce = ce_{safe}$. For the case of $ce = ce_{live} = ((x_0, \cdots, x_i), (x_{i+1}, \cdots, x_m))$, $sd = ((e_1, \cdots, e_q), (e_{q+1}, \cdots, e_k))$ is said to be orderly executed by t as by ce if t is in the form of $t = (x_0, x'_1, \cdots, x'_j)(x'_{j+1}, \cdots, x'_n)^\omega$ with $x'_n = x'_j$, and sd is orderly executed by $((x_0, x'_1, \cdots, x'_j), (x'_{j+1}, \cdots, x'_n))$ as defined above for the case of $ce = ce_{live}$.

In the above definition, the orderly execution of $sd = (e_1, \cdots, e_k)$ means that after the execution of e_j, e_{j+1} is the first one to be executed next among the events $\{e_j, \cdots, e_k\}$. Also note that in Definition 1, event labels are used to identify the transition executions instead of the state atomic proposition labels. This is because the event labels could help us to locate those transitions.

The fault seed sd defined in Definition 1 can be encoded by the LTL formula f_{sd} over the atomic proposition set Σ (i.e., each event $\sigma \in \Sigma$ is viewed as an atomic proposition), which is described below.

- For a safety specification, let $ce = ce_{safe} = (x_0, \cdots, x_m)$ and $sd = (e_1, \cdots, e_k)$, then f_{sd} is defined as:

  $$\begin{aligned} &f_{sd} = Ff_1 \\ &f_j = e_j \wedge X(\wedge_{r=j}^{k} \neg e_r U f_{j+1}),\ j = 1, \cdots, k-1 \\ &f_k = e_k \end{aligned}$$

- For a liveness specification, let $ce = ce_{live} = ((x_0, \cdots, x_i), (x_{i+1}, \cdots, x_m))$ and $sd = ((e_1, \cdots, e_q), (e_{q+1}, \cdots, e_k))$, then f_{sd} is defined as:

  $$\begin{aligned} &f_{sd} = Ff_1 \\ &f_j = e_j \wedge X(\wedge_{r=j}^{k} \neg e_r U f_{j+1}), 1 \leq j < k, j \neq q \\ &f_q = e_q \wedge GF f'_q \\ &f'_q = e_q \wedge X(\wedge_{r=q}^{k} \neg e_r U f_{q+1}) \\ &f_k = e_k \end{aligned}$$

From the semantics of LTL and Definition 1, we have the following result.

Theorem 1: Given a system M, a LTL specification S, a counterexample ce, and an event trace sd ($sd = (e_1, \cdots, e_k)$ if $ce = ce_{safe} = (x_0, \cdots, x_m)$; $sd = ((e_1, \cdots, e_q), (e_{q+1}, \cdots, e_k))$ if $ce = ce_{live} = ((x_0, \cdots, x_i), (x_{i+1}, \cdots, x_m))$), the following results hold:

1) sd is orderly executed by ce if and only if $t_{ce^\Sigma} \models f_{sd}$, where $t_{ce^\Sigma} = (\sigma_1 \cdots \sigma_m)\emptyset^\omega$ if $ce = ce_{safe} = (x_0, \cdots, x_m)$, and $t_{ce^\Sigma} = (\sigma_1 \cdots \sigma_i)(\sigma_{i+1} \cdots \sigma_m)^\omega$ if $ce = ce_{live} = ((x_0, \cdots, x_i), (x_{i+1}, \cdots, x_m))$ with $(\sigma_1, \cdots, \sigma_m)$ is the Σ event trace associated with $(x_1, \cdots, x_m)$, $(\sigma_1, \cdots, \sigma_i)$ and $(\sigma_{i+1}, \cdots, \sigma_m)$ are the Σ event traces associated with $(x_1, \cdots, x_i)$ and $(x_{i+1}, \cdots, x_m)$ respectively.
2) For any infinite trace t in the system, the following two statements are equivalent:
   - If sd is orderly executed by t as by ce, then t is faulty.
   - $M \models (f_{sd} \Rightarrow \neg S)$. More precisely, $\forall t \in T_M$, $t^{AP_\Sigma} \models (f_{sd} \Rightarrow \neg S)$.

From Theorem 1, we know that a fault seed can be identified by model checking. We have the following algorithm for the identification of a fault seed with a given length. Given a system M, a LTL specification S, a counterexample ce (either $ce = ce_{safe} = (x_0, \cdots, x_m)$ or $ce = ce_{live} = ((x_0, \cdots, x_i), (x_{i+1}, \cdots, x_m))$), a pair of integers (k_1, k_2) specifying the length of the fault seed to be identified in ce ($k_2 = 0$ for the case of $ce = ce_{safe}$), and let $(\sigma_1, \cdots, \sigma_m)$ be the Σ event trace associated with $(x_1, \cdots, x_m)$, then the algorithm for identifying a fault seed $sd = ((e_{i_1} \cdots e_{i_{k_1}})(e_{j_1} \cdots e_{j_{k_2}}))$ in ce is given below.

Algorithm 1: Algorithm for identifying a fault seed with the length (k_1, k_2)

1) Let $Candidate^{ce}_{(k_1,k_2)}$ be the set of candidates for the fault seeds with the length of (k_1, k_2) in ce, then initially we have $Candidate^{ce}_{(k_1,k_2)} = \{\sigma_j, j = 1, \cdots, m\}^{k_1}$ if $ce = ce_{safe}$, and $Candidate^{ce}_{(k_1,k_2)} = \{\sigma_j, j = 1, \cdots, i\}^{k_1}\{\sigma_j, j = i+1, \cdots, m\}^{k_2}$ if $ce = ce_{live}$.
2) If $Candidate^{ce}_{(k_1,k_2)} = \emptyset$ then stop the algorithm and output that no fault seed exists with the given length; otherwise pick a $sd \in Candidate^{ce}_{(k_1,k_2)}$ and let $Candidate^{ce}_{(k_1,k_2)} = Candidate^{ce}_{(k_1,k_2)} - \{sd\}$. Check whether $t_{ce^\Sigma} \models f_{sd}$. If the answer is yes then go to next step; otherwise repeat this step.
   To check whether $t_{ce^\Sigma} \models f_{sd}$, we first construct a system $M_1 = (X_1, R_1, x_1, \mathcal{L}_1, \mathcal{AP}_1)$ that can generate

the trace t_{ce^Σ}. M_1 is constructed as follows: $X_1 = \{x_1, \cdots, x_m, x_\epsilon\}$, $\mathcal{AP}_1 = \{\sigma_1, \cdots, \sigma_m\}$, $\mathcal{L}_1(x_\epsilon) = \emptyset$, $\mathcal{L}_1(x_j) = \{\sigma_j\}$ for $j = 1, \cdots, m$, $(x_j, x_{j+1}) \in R_1$ for $j = 1, \cdots, m-1$, $(x_m, x_{i+1}) \in R_1$ if $ce = ce_{live} = ((x_0, \cdots, x_i), (x_{i+1}, \cdots, x_m))$, $\{(x_m, x_\epsilon), (x_\epsilon, x_\epsilon)\} \subset R_1$ if $ce = ce_{safe} = (x_0, \cdots, x_m)$. It is not difficult to check that t_{ce^Σ} is the only $\mathcal{AP}_1$ trace generated by M_1. Next, $t_{ce^\Sigma} \models f_{sd}$ is the same as $M_1 \models f_{sd}$, i.e., model checking M_1 against the specification f_{sd}.

3) For the candidate sd, check whether $M \models (f_{sd} \Rightarrow \neg S)$, i.e., model checking M against the specification $(f_{sd} \Rightarrow \neg S)$. If the answer is yes then stop the algorithm and output sd as a fault seed with the length (k_1, k_2); otherwise go to Step 2.

Algorithm 1 can be used to find a fault seed sd with the length of (k_1, k_2) in a counterexample ce (if a fault seed with such a length exists). Since our goal is to find a fault seed in the counterexample irregardless of its length, we can use Algorithm 1 to search for a fault seed with different lengths exhaustively, i.e., trying all possible (k_1, k_2) such that $1 \leq k_1 + k_2 \leq |ce| - 1$, where $|ce|$ is the length of the counterexample ce. We can proceed from the shortest length of $k_1 + k_2 = 1$ to the longest length $k_1 + k_2 = |ce| - 1$ until we find a fault seed. With the observations that a long fault seed cannot provide much help for the fault explanation and in real applications the root cause of a fault usually can be identified by a limited number of transition executions, we can restrict the range of the lengths to be searched, i.e., we only search for all possible (k_1, k_2) with $1 \leq k_1 + k_2 \leq K$, where K is a reasonable small integer serving as the upper length bound of the fault seeds to be searched. Then the complexity of the above search for a fault seed with a length $1 \leq k_1 + k_2 \leq K$ is $O(|X|[min(|\Sigma|, |ce| - 1)]^K)$.

IV. An Illustrative Example

In this section we use an example to illustrate our approach for the fault explanation and isolation. We consider a communication application in the automotive CAN (controller area network) systems. An automotive CAN system consists of a number of ECUs (electrical control unit) that are connected by a CAN bus. In most cases, an ECU on the low-speed network with Single-Wire CAN is powered permanently. To save the battery power, such an ECU must enter a low power mode to reduce power consumption. Mostly, the *STOP* instruction of a micro-processor is used to enter a low power mode. But the *STOP* instruction must be executed in a way that an external interrupt can wakeup the micro-processor from this mode.

As identified in an application note ([7]), if a wakeup interrupt is used, which occurs only once after it has been enabled, the micro-processor may enter the *STOP* mode *after* this interrupt has been generated; and then there will be no further wakeup interrupt while the micro-processor is in *STOP* mode. In this case, the micro-processor will stay forever in the *STOP* mode, which is a malfunction of the system. Such a one-time wakeup interrupt is the wakeup interrupt from the CAN-controller, because the CAN-controller only generates the wakeup interrupt in the sleep-state and leaves immediately this state after a wakeup interrupt is generated. The explanation for the micro-processor could enter the *STOP* mode after the wakeup interrupt has been generated is given below as in [7]. When the application software on the ECU issues the *STOP* instruction to the micro-processor, the application software needs to enter a critical section to perform some non-interruptive tasks. After finishing those non-interruptive tasks, application software issues the *STOP* instruction to the micro-processor, and the micro-processor enters the *STOP* mode and stays there. So there is a short time window for the critical section. If the wakeup interrupt comes to the application during the short time window of the critical section, it will not be taken care of. As a result, the system continues to enter the *STOP* mode after the wakeup interrupt was generated. The reason for the above fault is that the wakeup interrupt comes while the system is in the critical section.

The above scenario is conceptually captured in Figure 1. There are four main parts in the system: *Application* on the

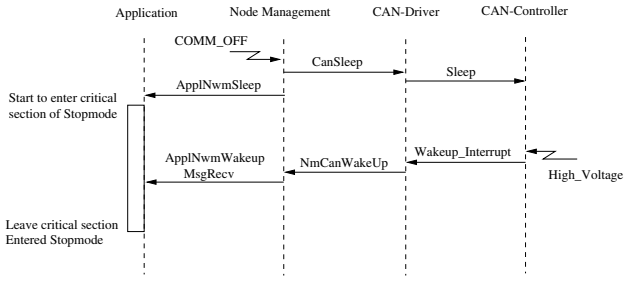

Fig. 1. A faulty scenario for the system

ECU to issue the *STOP* instruction, *Node Management* to manage the sleep and wakeup, *CAN-Driver* to interact with the *CAN-Controller*, and the *CAN-Controller* to connect to the CAN bus. The first three parts are software running on the micro-processor and the last part *CAN-Controller* is a hardware independent of the micro-processor. Suppose under certain conditions (such as no bus communication for some time), the system needs to go to sleep, which is represented as an input of *COMM_OFF* to the *Node Management*. Then the *Node Management* asks the *CAN-Driver* and the *Application* to go to sleep by calling the functions *CanSleep* and *ApplNwmSleep* respectively. Next, In the function call of *CanSleep*, the *CAN-Driver* asks the *CAN-Controller* to go to sleep by sending it *Sleep*; and in the function call of *ApplNwmSleep*, the *Application* asks the micro-processor to enter the *STOP* mode, and there is a critical section for entering the *STOP* mode. After receiving *Sleep*, the *CAN-Controller* goes to the sleep-state and could be waked up by a high level voltage from the bus when there is a need for bus communications. Once the *CAN-Controller* is waked up, it sends *Wakeup Interrupt* to the *CAN-Driver*; and then the *CAN-Driver* calls the function *NmCanWakeUp*, in which the *Node Management* calls *ApplNwmWakeupMsgRecv* to inform the *Application* of the wakeup. If the *Application* takes care of the interrupt then the micro-processor will be waked up.

However, as shown in Figure 1, the *ApplNwmWakeupMsgRecv* of the wakeup interrupt arrives while the *Application* is in the critical section, so the interrupt is not taken care of by the *Application* and the micro-processor continues to enter the *STOP* mode and stays there without any further *Wakeup Interrupt* to wake it up.

To apply our approach to the above example, we first need to model the system. The system is modeled in three modules: the *STOP-WakeUp Control* module, the *CAN-Controller* module, and the *Wakeup-Interrupt Handler* module, which are shown in Figure 2. The *STOP-WakeUp Control* module

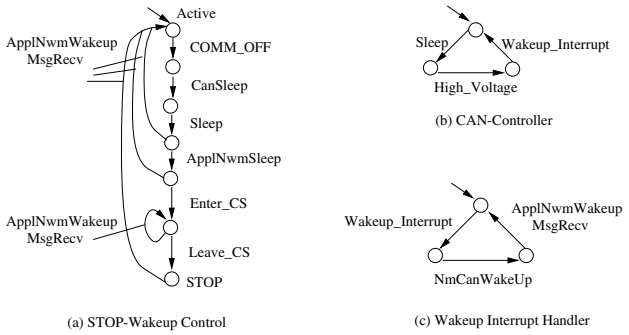

Fig. 2. Three modules in the system

combines the functions of *Application*, the *Node Management*, and the *CAN-Driver* for the control of entering the *STOP* mode and waking up to the active mode. The *CAN-Controller* module models the behavior of the CAN controller. The *Wakeup-Interrupt Handler* module models the handling of the *Wakeup Interrupt* from the *CAN-Controller*.

The model for the whole system can be obtained from the composition of the above three module models, which is shown in Figure 3, where *Active* is the only atomic proposition, which holds only at the initial state. In the

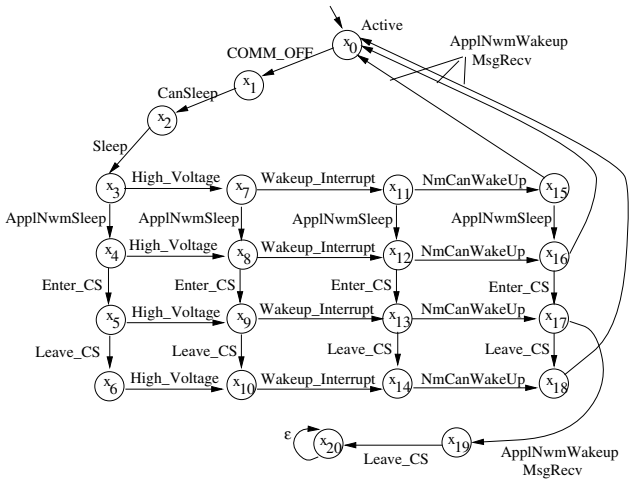

Fig. 3. The system model

model, there is self-loop labeled by ϵ at the state x_{20}. The ϵ is the null event and the loop is added to keep the system to be non-terminating. The specification of the system is given by the LTL formula "$GF\ Active$", which requires the system to be always able to return to active mode. Note that the specification represents a liveness property. The above system model can be transferred into a standard Kripke structure as described earlier, in which we basically remove the event label for each transition, add the event to the label of the destination state of the transition, and let the event be an atomic proposition that holds at the destination state of the transition. The detailed Kripke structure model is omitted here.

Next, we model check the system against the specification "$GF\ Active$". As expected, the system does not satisfy the specification and a counterexample is generated. Suppose the counterexample obtained is

$$\begin{aligned} ce \quad &= ce_{live} = (((x_0, \epsilon), (x_1, COMM\_OFF), \\ &(x_2, CanSleep), (x_3, Sleep), (x_4, ApplNwmSleep), \\ &(x_5, Enter\_CS), (x_9, High\_Voltage), \\ &(x_{13}, Wakeup\_Interrupt), (x_{17}, NmCanWakeUp), \\ &(x_{19}, ApplNwmWakeupMsgRecv), \\ &(x_{20}, Leave\_CS), (x_{20}, \epsilon)), ((x_{20}, \epsilon))) \end{aligned}$$

Now we can apply our approach to the above example. To find a fault seed for the above counterexample, Algorithm 1 is used. We use the length of the counterexample as the upper length bound for the fault seeds to be searched. Note that in the counterexample the first part contains 10 non-ϵ events *(COMM_OFF, CanSleep, Sleep, ApplNwmSleep, Enter_CS, High_Voltage, Wakeup_Interrupt, NmCanWakeUp, ApplNwmWakeupMsgRecv, Leave_CS)* and the second part does not contain any non-ϵ event. So we only need to consider the fault seeds of the length (k_1, k_2) with $k_1 \leq 10$ and $k_2 = 0$.

From Algorithm 1, we obtain the following fault seed for the counterexample: *sd=(Enter_CS, ApplNwmWakeupMsgRecv, Leave_CS)*. From the fault seed it is clear that the fault is caused by the coming of the *ApplNwmWakeupMsgRecv* between *Enter_CS* and *Leave_CS*, which means the wakeup interrupt comes while the application is in the critical section. Thus, our approach provides good fault explanation for the example.

V. Conclusion

In this paper, we studied the fault explanation and isolation problem for discrete event systems with LTL specifications. The notion of fault seed was introduced to characterized the cause of a fault. The fault seed was further encoded into LTL formula, and then the fault seed identification problem was reduced to a model checking problem. An algorithm based on model checking was developed for the fault seed identification. The effectiveness of the algorithm was demonstrated by an automotive communication example. For further research, one interesting topic would be to extend the approach to concurrent systems.

REFERENCES

[1] T. Ball, M. Naik, and S.K. Rajamani. From Symptom To Cause: Localizing Errors in Counterexample Traces. In *Proceedings of 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '03), New Orleans, LA, January 2003.

[2] S. Chaki, A. Groce, and O. Strichman. Explaining Abstract Counterexamples. In *Foundations of Software Engineering* (SIGSOFT FSE), pages: 73–82, Newport Beach, California, October-November 2004.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[4] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[5] A. Groce. Error Explanation with Distance Metrics. In *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), pages: 108–122, Barcelona, Spain, March-April 2004.

[6] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error Explanation with Distance Metrics. *International Journal on Software Tools for Technology Transfer*, accepted for publication.

[7] A. Happel. *Application Note: Enter Stopmode in a GMLAN environment*. Vector Informatik GmbH, 2002.

[8] S. Jiang and R. Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications. *IEEE Transactions on Automatic Control*, 49(6):934–945, 2004.