Automatic Model Generation for Black Box Real-Time Systems *

Thomas Huining Feng  
EECS, UC Berkeley  
tfeng@eecs.berkeley.edu

Lynn Wang  
EECS, UC Berkeley  
ting0918@eecs.berkeley.edu

Wei Zheng  
EECS, UC Berkeley  
zhengwei@eecs.berkeley.edu

Sri Kanajan  
General Motors  
sri.kanajan@gm.com

Sanjit A. Seshia  
EECS, UC Berkeley  
sseshia@eecs.berkeley.edu

Abstract

Embedded systems are often assembled from black box components. System-level analyses, including verification and timing analysis, typically assume the system description, such as RTL or source code, as an input. There is therefore a need to automatically generate formal models of black box components to facilitate analysis.

We propose a new method to generate models of real-time embedded systems based on machine learning from execution traces, under a given hypothesis about the system’s model of computation. Our technique is based on a novel formulation of the model generation problem as learning a dependency graph that indicates partial ordering between tasks. Tests based on an industry case study demonstrate that the learning algorithm can scale up and that the deduced system model accurately reflects dependencies between tasks in the original design. These dependencies help us formally prove properties of the system and also extract data dependencies that are not explicitly stated in the specifications of black box components.

1. Introduction

The design and verification of safety-critical real-time embedded systems involve the analysis of end-to-end latencies and task dependencies. This analysis requires having a precise and formal system model. However, in practice, many systems are assembled from black box components with imperfect accompanying specifications. In such situations, it is difficult to perform system integration and analysis without taking an extremely pessimistic view of the system. Automatic model generation mitigates this problem by providing a robust method of generating implicit dependencies and model features. The generated models facilitate verification of safety of real-time embedded systems. Natale et al. [10] provides a method of using this control flow.

As an instance, original equipment manufacturers (OEMs) in the automotive domain such as General Motors (GM) face many challenges related to the integration of electrical content [6], including the key issue of integrating multiple black box components into a single system. The OEMs tend to have a high level specification of the control flow model of a particular black box component, but when the components are integrated, the system level control flow model is difficult to infer especially in the presence of non-determinism from the operating system [1] and the CAN communication bus [3]. Hence, performing an end-to-end timing analysis is difficult without assuming that all messages and tasks are potentially independent at the system level [13]. This approach is extremely pessimistic.

To improve end-to-end analysis, we present a novel machine learning-based approach to automatically generate a system-level control flow model from execution traces of real-time embedded systems. Past work includes model generation based on iterative processes on recording real-time execution traces [5]. This method is high in complexity. There also exist techniques for generating a model for finite-state systems by observing execution traces based on a machine learning algorithm first proposed by Angluin [2] and improved upon by Rivest and Schapire [12]. However, techniques are not well-defined for real-time systems, not even for learning partial orders between tasks and events. Our formulation of model generation as the learning problem is inspired by the work of Lau et al [7] on programming by demonstration. To our knowledge, ours is the first work on automatic generation of a real-time control flow model from execution traces. Unlike methods that statically analyze the model’s design, our method analyzes the system’s execution traces without requiring to know the design, thus identifying not only dependencies intended in the design but also unintended dependencies introduced by the execution environment. This helps to detect more dependencies that

---

*This research was supported in part by the MARCO Gigascale Systems Research Center and CHESS (the Center for Hybrid and Embedded Software Systems), which receives support from NSF and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.
Figure 1. A simple system design model

Figure 2. An example trace with three periods

Figure 1 is derived from a system design model, where tasks are illustrated as nodes and messages as edges. The edges represent dependencies between tasks. Figure 2 shows an example trace with three periods, where each period contains a sequence of messages between tasks.

Actually, we do not have access to the system design. Instead, we try to reconstruct dependency models, whose edges represent dependencies. This type of model is different from the system design, because a task may indirectly influence another with no explicit messages between them. In this paper, we refer to messages as messages in system designs; we refer to dependencies as dependencies in dependency models.

We distinguish two types of nodes. A disjunction node is one that conditionally sends messages to other tasks, and in this way chooses execution paths, such as $t_1$ above. A conjunction node is one that passively receives messages from other tasks, depending on the decisions that others made, e.g., $t_4$. We further assume that in any period, there could be at most one message sent between any sender-receiver pair. For example, if $t_1$ intends to send 2 pieces of data to $t_2$ in a period, it groups the data and sends them in one message. This is realistic because we assume that messages can only be sent when the sender task finishes. Thus, the overhead is reduced by grouping the data at the sender side and sending them all at once.

A trace is a timestamped sequence of events, where an event is the start or end of a task, or the rising edge or the falling edge of a message transmitted on the bus. From GM, the trace is obtained with a logging device connected to the communication bus shared by all tasks. The bus gives us no information about the senders and the receivers of the messages. Also, we do not know a priori the meaning of the messages, or whether a node is disjunction, or conjunction, or neither of the two.

Figure 2 is an imaginary execution trace of the above example. In a period, a task may execute at most once. A task can not execute if it does not receive its required messages. If executed.

2. The Learning Problem

2.1. Model of Computation

A model of computation (MOC) is the abstraction of a system into a model on which one could do mathematical computations [8]. The MOC assumed here is a control flow MOC [11]. The basic rule is that tasks are executed in a data driven manner where the firing rule of the task is the arrival of all its required inputs.

An automotive system is modeled with a set of predefined tasks repeatedly being executed in periods. After a task completes execution, it may send messages to other tasks to be executed in the same period. We assume that no message may cross the period boundary.

The system can be represented with a graph, which defines all possible behavior within one single period. Different periods in an execution conform to the same graph, although the behavior need not be identical due to the logical decisions made. Nodes in the graph are individual tasks. The edges represent messages between tasks. Figure 1 shows an example of this type of model, where $t_1$ is designed to send message(s) to $t_2$ or $t_3$ or both in each period, and $t_2$ and $t_3$ independently send messages to $t_4$ if executed.

However, we assume that we do not have access to the system design. Instead, we try to reconstruct dependency models, whose edges represent dependencies. This type of model is different from the system design, because a task may indirectly influence another with no explicit messages between them. In this paper, we refer to messages as messages in system designs; we refer to dependencies as dependencies in dependency models.

We distinguish two types of nodes. A disjunction node is one that conditionally sends messages to other tasks, and in this way chooses execution paths, such as $t_1$ above. A conjunction node is one that passively receives messages from other tasks, depending on the decisions that others made, e.g., $t_4$. We further assume that in any period, there could be at most one message sent between any sender-receiver pair. For example, if $t_1$ intends to send 2 pieces of data to $t_2$ in a period, it groups the data and sends them in one message. This is realistic because we assume that messages can only be sent when the sender task finishes. Thus, the overhead is reduced by grouping the data at the sender side and sending them all at once.

A trace is a timestamped sequence of events, where an event is the start or end of a task, or the rising edge or the falling edge of a message transmitted on the bus. From GM, the trace is obtained with a logging device connected to the communication bus shared by all tasks. The bus gives us no information about the senders and the receivers of the messages. Also, we do not know a priori the meaning of the messages, or whether a node is disjunction, or conjunction, or neither of the two.

Figure 2 is an imaginary execution trace of the above example. In a period, a task may execute at most once. A task can not execute if it does not receive its required messages.

2.2. Basic Definitions

The following basic definitions are due to Mitchell [9]:

Definition 1 (Instances). $I$ is the set of instances for the learning problem. In this work, we do not consider negative examples.

Definition 2 (Hypothesis space). $H$ is the hypothesis space. A partial order (see below) is defined on this space. Each $h \in H$ is an approximation to the desired property with respect to the partial order.

Definition 3 (Matching function). $M : H \times I \rightarrow \text{boolean}$ is the matching function. $M(h,i)$ for $h \in H$ and $i \in I$ is true if and only if hypothesis $h$ matches instance $i$. We may also write $M : H \times \mathcal{P}(I) \rightarrow \text{boolean}$, so that $\forall I_0 \subseteq I, (M(h,i) \iff \forall i \in I_0, M(h,i))$.  \(^1\)

Definition 4 (More-specific-than relation). The partial order $\sqsubseteq_H$ on $H$ is defined in terms of more-specific-than relation: $\forall h_1, h_2 \in H$, $h_1$ is more specific than $h_2$ if and only if $\forall i \in I, M(h_1,i) \Rightarrow M(h_2,i)$. This is denoted by $h_1 \sqsubseteq_H h_2$. ($\sqsubseteq_H$ is defined similarly.)

$I$ is an execution trace. Each instance $i \in I$ is a period in that trace (order irrelevant). $H$ is the set of hypotheses about task dependencies. $M$ indicates whether a hypothesis

\(^1\) $\mathcal{P}(I)$ represents the power set of $I$. 
Definition 6 (Abstracted learning problem). Given \( I \), with \( T \), \( D, \subseteq D \), and \( M \) predefined, find \( D, \subseteq D \) such that
\[ \forall d \in D. M(d, I) \Rightarrow (\exists d^* \in D^*, d^* \subseteq D, d). \] This is the completeness and optimality requirement. \(^2\)

The above abstract learning problem is independent of how the lattice of hypotheses is formed. In this paper, we assume \( D, \subseteq D \) to be given in Definition 5, which fits the envisioned applications. For other applications, different lattices of hypotheses may be defined. The algorithm below is still applicable. However, the result varies depending on the lattice used. This is consistent with the fact that in practice, the same trace may be generated by different concrete models (each of which is the most specific result in its lattice).

3. The Generalization Algorithm

The input to the algorithm is an exhaustive trace of time-stamped events. Our algorithm analyzes the trace starting with the set containing only the most specific hypothesis. Every time a new instance is given, the algorithm tries to match it with the hypotheses in that set. Hypotheses that do not match will be generalized.

3.1. Message-Guided Generalization

Starting from \( D_0 = \{d_\perp\} \) with \( d_\perp \) being the globally most specific hypothesis, the algorithm handles one period in the trace at a time. This learning process is incremental. The current set of hypotheses keeps growing in terms of generality but not necessarily cardinality.

We denote the \( k \) occurrences of messages in the trace with \( m_1, m_2, \ldots, m_k \). Since each period corresponds to an instance, we denote periods with \( t_1, t_2, \ldots, t_n \). If \( m_p \) belongs to \( i_q (1 \leq p \leq k, 1 \leq q \leq n) \), we write \( m_p \preceq i_q \).

Initially, when the algorithm is provided with \( i_1 \), it first analyzes the first message in it, i.e., \( m_1 \preceq m_1 \). Its possible sender-receiver pairs are computed: \( A_{m_1} = \{(s, r) \in T \text{ can be } m_1 \text{'s sender } \land r \in T \text{ can be } m_1 \text{'s receiver}\} \). Then for any sender-receiver combination \( (s_1, r_1) \in A_{m_1} \), we create a hypothesis \( d_{11} \) by generalizing from \( d_\perp \) with this assumption, such that \( \forall t_1, t_2 \in T \):

\[
d_{11}(t_1, t_2) = \begin{cases} d \land t_1 = s_{11} \land t_2 = r_{11}, \\ t_1 = r_{11} \land t_2 = s_{11}, \\ d \land t_1 = s_{11}. \
\end{cases}
\]

Note that each time we only generalize as much as necessary. For example, we could let \( d_{11}(t_1, t_2) \) be \( \rightarrow \) instead of \( \Rightarrow \) as above, and it still satisfies the correctness requirement, but it does not satisfy the optimality requirement.

With \( n_1 \) different assumptions of \( m_1 \ (n_1 = |A_{m_1}|) \), a set of new hypotheses is obtained: \( D_1 = \{d_{11}, d_{12}, \ldots, d_{1m_1}\} \).

If \( m_2 \) also is in \( i_1 \ (m_2 \preceq i_1) \), the algorithm also analyzes \( m_2 \) after analyzing \( m_1 \). It also computes the

\(^2\)The reason for finding the most specific hypotheses is that any more general hypothesis will automatically match all the instances when no negative instance exists.
set of possible sender-receiver pairs: $A_{m_2} = \{(s,r)| s \in T \text{ can be } m_2 \text{'s sender } \land r \in T \text{ can be } m_2 \text{'s receiver}\}$. Then it tries to generalize the hypotheses in $D_1$ for any assumed sender-receiver pair (if necessary). For any $d_{ij} \in D_1$, we need to find the set $D_{1j} = \{d_{1j1, d_{1j2}, \ldots, d_{1jm}}\}$ such that for any $d_{ijk} \in D_{1j}$, the following conditions hold:

1. $d_{ijk} \subseteq D_{1j}$
2. $M(d_{ijk}, D_{1j}) = \text{true}$
3. The sender-receiver pair $(s, r)$ that $d_{ijk}$ assumes is not in the assumptions of $d_{ij}$. As mentioned above, we assume that between any two data dependent tasks, there can be only one message between them in a period.
4. For the optimality requirement, $\neg \exists d' \in D \text{ s.t. } d_j \subseteq D d' \subseteq D \text{ and } d' \text{ still satisfies the above conditions.}$

This means we generalize only as much as necessary. When it is obtained from $d_{ij}$, $d_{ijk}$ will have all the assumptions that $d_{ij}$ has, plus one new assumption of the sender-receiver pair for $m_2$. As a result, after analyzing $m_2$, we obtain $D_2 = \bigcup D_{1j}$.

The assumptions help to efficiently reduce the number of hypotheses. Our system assumes that in any period, for any sender-receiver pair $(s, r)$, there can be at most one message from $s$ to $r$. If a hypothesis was obtained earlier by assuming $s$ to send a message to $r$, then later in the same period, we will not consider the same sender-receiver pair.

The algorithm repeats until all the messages in $t_1$ are analyzed. At the end of the period, a post-processing operation is performed. The post-processing operation first removes the assumptions associated with the hypotheses. Two or more hypotheses in the current set $D_{cur}$ may become equal and thus be unified. The post-processing operation also removes redundant hypotheses. $d \in D_{cur}$ is redundant if and only if $\exists d' \in D_{cur}, d' \subseteq D d$. This is because 1) all the hypotheses in $D_{cur}$ match the instances seen so far, and 2) we are trying to find the most specific hypotheses (with respect to $\subseteq D$).

The learning then continues with $t_2$, until the entire trace is analyzed. If $D_{cur}$ becomes $\emptyset$ at some point, it means 1) either the instances contain errors (and thereby violate our assumptions), or 2) the generalization language is not expressive enough to describe the desired property. If only one hypothesis is left at the end, we say that the algorithm converges to a unique most specific solution. If two or more hypotheses are left, more periods in the trace are needed to reveal other aspects of the model, and make the algorithm converge.

### 3.2. Heuristics

The algorithm discussed above is exponential in the number of messages. Hardness of this problem will be

---

3It may not be possible to perceive every aspect of the model’s behavior because of the scheduler’s property. For example, the scheduler for the execution may produce deterministic schedules though the model allows non-determinism. Therefore, the execution may always exhibit a fixed part of the model’s allowed behavior, and the dependency functions learned will be more specific than the dependencies intended in the model’s design.

---

proved by Theorem 1 in Section 4. We develop a heuristic which does not compromise the algorithm’s soundness. However, it is conservative because the result is no longer guaranteed to be the most specific. We will justify this conservativeness with the convergence theorem.

We keep an ordered list of current hypotheses, instead of the unordered set $D_{cur}$ in the previous algorithm. A weight function is used as the ordering criterion. This particular weight function is used to make all dependencies in $D$ inter-comparable. As in Figure 3 a parallel execution is more specific than a directed execution, and that is more specific than a probable dependency. The higher a dependency-relation is in the lattice, the more weight we give to that hypothesis.

**Definition 7 (Distance).** $\text{distance} : V \rightarrow \mathbb{N}$ computes the square distance (a natural number) from any dependency value to the lattice bottom $\|\|$:

$$\text{distance}(v) = \begin{cases} 0, & v \in \{\|\}\} \\ 1, & v \in \{\leftarrow, \rightarrow\} \\ 2, & v \in \{\leftarrow, \leftrightarrow, \rightarrow\} \\ 3, & v \in \{\leftarrow, \leftrightarrow\} \end{cases}$$

**Definition 8 (Weight).** $\text{weight} : D \rightarrow \mathbb{N}$ is defined as

$$\text{weight}(d) = \sum_{t_1, t_2 \in T} \text{distance}(d(t_1, t_2))$$

Hypotheses are ordered by the weight function in the list. According to the heuristics, every time a new hypothesis is added, if the total number of hypotheses becomes 1-greater than a given bound, the two hypotheses with the least weights are replaced with their least upper bound.

### 3.3. A Simple Example

We will demonstrate the generalization algorithm with the example introduced in Section 2. After analyzing $m_1$, there are two most specific hypotheses:

<table>
<thead>
<tr>
<th>$d_{11}$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$d_{12}$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$t_2$</td>
<td>$t_3$</td>
<td>$t_4$</td>
<td>$t_1$</td>
<td>$t_2$</td>
<td>$t_3$</td>
<td>$t_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_1$</td>
<td>$t_2$</td>
<td>$t_3$</td>
<td>$t_4$</td>
<td>$t_1$</td>
<td>$t_2$</td>
<td>$t_3$</td>
<td>$t_4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The assumptions of the hypotheses are shown below the tables. $d_{11}$ is obtained by assuming $m_1$ to be sent from $t_1$ to $t_2$, while $d_{12}$ assumes $m_1$ to be from $t_1$ to $t_4$. After this step, the current set of hypotheses $D_{cur} = \{d_{11}, d_{12}\}$. Both $d_{11}$ and $d_{12}$ are the most specific hypotheses, and they are correct with respect to the first message.

The algorithm further handles $m_2$ by generalizing any hypothesis in $D_{cur}$ (if necessary). New hypotheses with duplicated assumptions will not be considered. For example, $d_{12}$ assumes $m_1$ to be sent from $t_1$ to $t_4$. This assumption will not allow us to create a new hypothesis with an assumption about $m_2$ being sent from $t_1$ to $t_4$. The following tables show the three new hypotheses that we obtain:
After period 1, we update $D_{car}$ with $\{d_{21}, d_{22}, d_{23}\}$. Post-processing operations are performed at the end of each period to remove all the assumptions, to test conditional dependencies, and to delete redundant hypotheses.

The algorithm then proceeds to period 2 and then period 3. After period 3, these 5 hypotheses remain in $D_{car}$:

### $d_{31}$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

### $d_{32}$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

### $d_{33}$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

### $d_{34}$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

### $d_{35}$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

These hypotheses are the most specific ones that satisfy all the instances. However, because of the limited number of instances, the algorithm does not converge. We may consider their least upper bound $d_{LUB}$ (which no longer guarantees optimality) as the final result:

### $d_{LUB}$

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

$d_{LUB}$ satisfies $\forall s \in T, t \in T.d_{LUB}(s, t) = d_{31}(s, t) \sqcup d_{32}(s, t) \sqcup \cdots \sqcup d_{35}(s, t)$, where $\sqcup$ is least upper bound operator on $V$, defined by the lattice in Figure 3.

The result is illustrated in Figure 4. One interesting result is: $t_1$ always determines $t_4 (\rightarrow)$. This result cannot be acquired by merely looking at the original model (even if we have it). With the original model, we could only tell that $t_1$ may or may not send messages to $t_2$ and $t_3$, but we did not have this unconditional dependency by transitive closure.

### 3.4. Case Study

The algorithm was applied to a distributed system comprised of 18 tasks and 330 messages transmitted on one CAN bus. The execution trace contained 27 periods and 700 event-pair executions of tasks and messages.

The original model was a General Motors (GM) controller in a black box. For proprietary reasons, we cannot disclose actual names of tasks. We abstract these tasks using letters A to P and S. Heuristics were used to reduce runtime. Results in the textual format were translated into a dependency graph (Figure 5). We used this dependency graph to prove properties (e.g., dependencies and operation mode of tasks) of the system, assuming that the trace is exhaustive so that it exhibits all allowable behavior of the model in the specific execution environment. The output of the algorithm confirmed some properties that were known in advance; e.g. Tasks A and B are disjunction nodes. Other properties are learned, e.g. Tasks H, P and Q are conjunction nodes, no matter which mode task $A$ chooses, task $L$ must execute $(d(A, L) = \rightarrow)$, and no matter which mode task $B$ chooses, task $M$ must execute $(d(B, M) = \rightarrow)$.

High-level properties such as “if the brake is pressed, then brake actuator must react within 300 msec,” are known from the specification. However, some properties are not known at design time; for example the data dependency between Q and O (Figure 5) comes from the interactions between the functional tasks and the infrastructure tasks, namely the CAN bus scheduler and the OSEK scheduler.

The additional dependencies discovered from the execu-
tion trace help to reduce the state space that needs to be analyzed with other methods. One such method could be model checking by means of reachability analysis. Reduced state space results in more efficient model checking, and less false alarms produced.

The dependency relations that we obtained also significantly improve the pessimistic analysis of end-to-end latencies of the original system model. For example, one path that was examined in this case study was the critical path including task Q. Our learning algorithm introduces an implicit dependency between task Q and O, which is less pessimistic when calculating the end-to-end path latency in the way of excluding the possible preemption from higher priority task O during the execution of task Q.

Our implementation was tested with different bounds on a Windows XP machine with a Pentium M 1.7 GHz processor and 1 GB memory, as shown in the following table.

<table>
<thead>
<tr>
<th>Bound</th>
<th>Run time (sec)</th>
<th>Bound</th>
<th>Run time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.220</td>
<td>64</td>
<td>5.899</td>
</tr>
<tr>
<td>4</td>
<td>0.471</td>
<td>100</td>
<td>12.608</td>
</tr>
<tr>
<td>16</td>
<td>1.202</td>
<td>120</td>
<td>16.294</td>
</tr>
<tr>
<td>32</td>
<td>2.573</td>
<td>150</td>
<td>19.048</td>
</tr>
</tbody>
</table>

We also experimented with the precise but exponential algorithm. It took 630.997 seconds and returned a single dependency function, which equaled the least upper bound of the dependency functions we obtained with heuristics (using any arbitrary bound). Theorem 4 in the next section will show that this equality is not a coincidence.

4. Properties of the Algorithm

**Theorem 1 (NP-hardness).** The problem of finding the set of most specific hypotheses for a given trace is NP-hard.

**Theorem 2 (Correctness).** The algorithm (with or without heuristics) guarantees correctness. If the algorithm returns $D^*$ as the set of hypotheses, $v d^* \in D^*.M(d^*, I)$.

**Theorem 3 (Optimality and completeness).** The algorithm without heuristics guarantees optimality and completeness. If the algorithm returns $D^*$ as the set of hypotheses, then $\forall d \in D.M(d, I) \Rightarrow (\exists d^* \in D^*.d^* \subseteq B.d)$.

**Lemma.** If the algorithm returns $D^*$ when the bound is set to $b$, and if $d^*$ is the hypothesis obtained with the bound set to 1, then $d^* = \bigcup D^*$ (the least upper bound of all the elements in $D^*$).

**Theorem 4 (Convergence).** If the algorithm converges to one hypothesis $d^*_1$, regardless of whether the bound is set or what the bound is, and if the algorithm returns $d^*_2$ with the bound set to 1, then $d^*_1 = d^*_2$.

Theorem 1 is proved by transforming the Boolean Satisfiability Problem (SAT). Theorem 2 and Theorem 3 are proved by induction on periods. Theorem 4 is a direct consequence of the lemma, which is due to the fact that there is a unique least upper bound for any two elements in a lattice. The proofs can be found in a technical report [4].

The runtime of heuristics-based algorithm is $O(mb^2 + mb^2 t)$, where $m$ is the number of messages in the trace, $t$ is the number of tasks, and $b$ is the user-specified bound.

5. Conclusion

We designed and implemented an algorithm that constructs a dependency graph from execution traces using machine learning techniques based on generalization of hypotheses. The algorithm is proved correct and optimal (in terms of the lattice that we define). The algorithm terminates in polynomial time (in the number of tasks, the number of messages and the bound) using heuristic search.

Though the work’s motivation originated from an automotive application, the algorithm could be applied elsewhere. It could also be extended by version space techniques provided negative examples in the execution traces.

References