Automatic Model Generation for Black Box Real-Time Systems

Huining Thomas Feng
Lynn Tao-Ning Wang
Wei Zheng
Sri Kanajan
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-117

http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-117.html

September 25, 2006

Acknowledgement

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.

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, thereby facilitating verification of safety of real-time embedded systems.

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 [5], 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 nondeterminism 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 [11]. This approach is extremely pessimistic. Instead, by automatically generating the system-level control flow model defining actual data dependencies between tasks, this end-to-end timing analysis pessimism is significantly improved.

In this paper, 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 include model generation based on iterative processes on recording real-time execution traces, but this method is high in complexity [4]. Our formulation of model generation as the learning problem is inspired by the work of Lau et al [6] on programming by demonstration. There already exist techniques for automatically 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 [10]. However, techniques are not well-defined for real-time systems, including for learning partial orders

[*] 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.

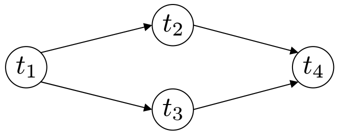

Figure 1. A simple system design model

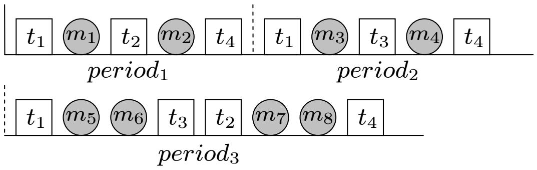

Figure 2. An example trace with three periods

between tasks and events. To our knowledge, ours is the first work on automatic generation of a real-time control flow model from execution traces. We give two algorithms: one that is optimal, but worst-case exponential-time, and the other that is heuristic, but converges to a learned model in polynomial time. We demonstrate the practical applicability of the latter algorithm by applying it to an industrial case study from GM.

2. The Learning Problem

In this section, we present the underlying formalism including the formulation of model generation as a 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 [7] . The MOC assumed here is a control flow MOC [9]. 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 ends executing, 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 unique 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 are trying to reconstruct *dependency models*, whose edges represent dependencies. This type of model is different from the system design model mentioned previously, because a task may indirectly depend on or determine another with no explicit messages between them. When we mention messages, we are talking about system designs; when we mention dependencies, we are assuming dependency models.

We distinguish two special types of nodes. A *disjunction node* is one that conditionally sends messages to other tasks, and in this way choose 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. E.g., if t_1 wants to send 2 events to t_2 in a period, it groups the events and send them in one message. This is realistic because we assume that messages can only be sent when the sender task finishes. Thus, the overhead of sending multiple events is reduced by queuing the events at the sender side and sending them all at once.

A trace is a time stamped sequence of events, where an event is the start or end of a task or message. However, we have no information about the sender, the receiver, or the contents of a message. Nor do we assume to know a priori whether a node is disjunction, or conjunction, or neither of the two.

Figure 2 is an imaginary execution trace of the above example. It shows only 3 periods. In a period, a task may execute at most once. A task can not execute if it does not receive its required message(s).

2.2. Basic Definitions

The following basic definitions are due to Mitchell [8]:

Definition 1 (Instances). I is the set of instances for the learning problem. In our case, each instance $i \in I$ represents a fact, i.e. no 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). $M : H \times I \rightarrow boolean$ 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. Informally, we may as well use $M : H \times \mathcal{P}(I) \rightarrow boolean$ depending on the context, so that $\forall I_0 \subseteq I.(M(h,I_0) \Leftrightarrow \forall i \in I_0.M(h,i))$. [1]

Definition 4 (More-specific-than relation). The partial order $\sqsubseteq_H$ on H is defined in terms of more-specific-than re-

[1] $\mathcal{P}(I)$ represents the power set of I.

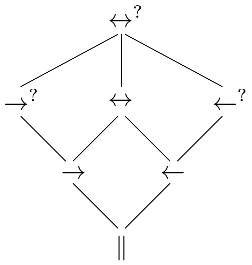

Figure 3. Lattice of dependency values

lation: $\forall h_1, h_2 \in H$, 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 $h_1 \sqsubseteq_H h2$. ($\sqsubset_H$ is defined similarly.)

In our case, I is an execution trace. Each instance $i \in I$ is a period in that trace (order irrelevant). H is the set of hypotheses of task dependencies. M indicates whether a hypothesis matches an instance. By "matching" we mean that the instance period conforms to the hypothesized dependency. E.g., if the instance contains a message assumed to be sent from s to r (an example later shows how assumptions are made), any hypothesis that matches this instance should define a directed dependency between s and r.

2.3. Problem Formulation

We formulate dependencies with functions. A partial order is defined on the set of possible dependency functions.

Definition 5 (Dependency functions). $d \in D : T \times T \to V$, where T is the set of predefined tasks, and $V = \{\|, \rightarrow, \leftarrow, \leftrightarrow, \rightarrow^?, \leftarrow^?, \leftrightarrow^?\}$ is the set of possible dependency values. For any $t_1, t_2 \in T$:

- $d(t_1, t_2) = \|$: t_1 always executes in parallel with t_2.
- $d(t_1, t_2) = \rightarrow$: if t_1 executes in a period, it always determines the execution of t_2.
- $d(t_1, t_2) = \leftarrow$: if t_1 executes in a period, it always depends on the execution of t_2.
- $d(t_1, t_2) = \leftrightarrow$: t_1 and t_2 depend on each other. (This relation never happens in our case. It is defined only for completeness.)
- $d(t_1, t_2) = \rightarrow^?$: if t_1 executes in a period, it may or may not determine the execution of t_2.
- $d(t_1, t_2) = \leftarrow^?$: if t_1 executes in a period, it may or may not depend on the execution of t_2.
- $d(t_1, t_2) = \leftrightarrow^?$: t_1 and t_2 may or may not depend on/determine each other.

We define a partial order $\sqsubseteq_V$ on V. This partial order is illustrated in Figure 3 in the form of a lattice.

We define a partial order $\sqsubseteq_D$ on D. $\forall d_1, d_2 \in D . (d_1 \sqsubseteq_D d_2 \Leftrightarrow \forall t_1, t_2 \in T . d_1(t_1, t_2) \sqsubseteq_V d_2(t_1, t_2))$. $\sqsubseteq_D$ is also a lattice.

We further define the most specific hypothesis $d_\bot \in D$ so that $\forall t_1, t_2 \in T . d_\bot(t_1, t_2) = \|$, and the least specific hypothesis $d_\top \in D$ so that $\forall t_1, t_2 \in T . d_\top(t_1, t_2) = \leftrightarrow^?$. Obviously, $\forall d \in D . d_\bot \sqsubseteq_D d \sqsubseteq_D d_\top$.

In this paper, hypotheses are dependency functions, so we define hypothesis space $\langle H, \sqsubseteq_H \rangle$ with $H ::= D$ and $\sqsubseteq_H ::= \sqsubseteq_D$.

Definition 6 (Abstracted learning problem). Given I, with T, $\langle D, \sqsubseteq_D \rangle$, and M predefined, find $D^* \subseteq D$ such that

1. $\forall d^* \in D^* . M(d^*, I)$. This is the correctness requirement.
2. $\forall d \in D . M(d, I) \Rightarrow (\exists d^* \in D^* . d^* \sqsubseteq_D d)$. This is the completeness and optimality requirement. [2]

3. The Generalization Algorithm

The input to the algorithm is an exhaustive trace of message and task executions along with a global timestamp. Our algorithm is based on breadth-first-search and additional heuristics to bound memory consumption. It starts 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 the new instance will be generalized.

3.1. Message-Guided Generalization

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

We denote the k occurrences of messages in the trace with $m_1, m_2, \cdots, m_k$. Since each period corresponds to an instance, we denote periods with $i_1, i_2, \cdots, i_n$. Multiple messages may be found in a single period. If m_p belongs to i_q, we write $m_p \propto i_q$.

Initially, when the algorithm is provided with i_1, it first analyzes the first message in it, which is $m_1 \propto i_1$. Its possible sender-receiver pairs are computed: $\{(s, r) | s \in T \text{ can be the sender of } m_1 \wedge r \in T \text{ can be the receiver of } m_1\}$. Then for any sender-receiver combination, we create a new hypothesis with this combination as its assumption. E.g., if s and r are assumed to be the sender and receiver, respectively, we may generalize $d_\bot$ to d_{11} such that $\forall t_1, t_2 \in T$:

$$d_{11}(t_1, t_2) = \begin{cases} \rightarrow & t_1 = s \wedge t_2 = r \\ \leftarrow & t_1 = r \wedge t_2 = s \\ \| & \text{otherwise} \end{cases}$$

[2] The reason for finding the most specific hypotheses is that any more general dependency function will automatically match all the instances.

Note that each time we only generalize as much as necessary. E.g., we could let $d_{11}(s,t)$ be $\rightarrow^?$ instead of $\rightarrow$ as defined above, and still satisfy the correctness requirement. However, this is not the most specific generalization.

With n_1 different assumptions of m_1, a set of new hypotheses is obtained: $D_1 = \{d_{11}, d_{12}, \cdots, d_{1n_1}\}$.

If m_2 is also in i_1 ($m_2 \propto i_1$), the algorithm also analyzes m_2 after analyzing m_1. It also computes the set of possible sender-receiver pairs: $\{(s,r) | s \in T \text{ can be the sender of } m_2 \wedge r \in T \text{ can be the receiver of } m_2\}$. Then we try to generalize hypotheses in D_1 for any assumed sender-receiver pair (if necessary). For any $d_{1j} \in D_1$, we try to find the set $D_{1j} = \{d_{1j1}, d_{1j2}, \cdots, d_{1jm}\}$ such that for any $d_{1jk} \in D_{1j}$, the following conditions hold:

1. $d_{1j} \sqsubseteq_D d_{1jk}$
2. $M(d_{1jk}, i_2) = true$
3. The sender-receiver pair (s,r) that d_{1jk} assumes is not in the assumptions of d_{1j}. As mentioned above, we assume that between any two data dependent tasks, there can be only one message between them in a period.
4. The optimality requirement: $\neg\exists d' \in D$ s.t. $d_{1j} \sqsubset_D d' \sqsubset_D d_{1jk}$ and d' still satisfies the above conditions (because we generalize only as much as necessary).

When it is obtained from d_{1j}, d_{1jk} will have all the assumptions that d_{1j} has, plus one new assumption of the sender-receiver pair for m_2.

The assumptions are important because they help to efficiently reduce the number of hypotheses. The system we defined in this paper 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 obtained earlier already assumes s to send a message to r, then later in the same period, we will not consider the same sender-receiver pair again.

The algorithm repeats this until all the messages in i_1 are analyzed. At the end of the period, a post-processing operation is performed. The post-processing operation first deletes the assumptions of hypotheses in the set. Two or more hypotheses in the current set D_{cur} may become equal and thus be unified. The post-processing operation also tries to shrink D_{cur} by removing redundant hypotheses. $d \in D_{cur}$ is redundant if and only if $\exists d' \in D_{cur}.d' \sqsubset_D d$. This means, if d is strictly more general than any other hypothesis, then it can be removed. 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 (w.r.t. $\sqsubseteq_D$).

The algorithm processes all the periods in the same way, until the whole trace is learned. If $\varnothing$ is obtained at the end, it means either the instances contain errors, or the generalization language is not expressive enough to describe the desired property. If only one hypothesis is left in the set, we say the algorithm *converges* to a unique most specific solution. If there are two or more hypotheses left, we need more traces that reveal other aspects of the model. [3]

3.2. Heuristics

The algorithm discussed above is exponential in the number of messages. Hardness of this problem is proved by Theorem 1. We develop a heuristic which does not violate the algorithm's soundness. However, it is *conservative* because the result is no longer guaranteed to be the most specific. This conservativeness is later justified by the convergence theorem.

Instead of keeping a set of current hypotheses, we keep an ordered list of them. A *weight* function is used as ordering criteria. 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. Naturally, the higher a dependency-relation ranks in the lattice height, the more weight we give to that hypothesis.

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

$$distance(v) = \begin{cases} 0, & v \in \{||\} \\ 1, & v \in \{\rightarrow, \leftarrow\} \\ 4, & v \in \{\rightarrow^?, \leftrightarrow, \leftarrow^?\} \\ 9, & v \in \{\leftrightarrow^?\} \end{cases}$$

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

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

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 the 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 in Section 2. There are 4 tasks in total. Figure 2 illustrates the first three periods of a possible trace. After analyzing m_1, there are two most specific hypotheses:

[3] This may not be possible because of the scheduler's property. The scheduler used in the model's execution may not produce strictly random schedules that the model allows. I.e., it may always exhibit a fixed part of the model's possible behavior, so the dependency functions learned from the trace will be more specific than the real dependency function.

d_{11}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow$	$\parallel$	$\parallel$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\parallel$
t_3	$\parallel$	$\parallel$	$\parallel$	$\parallel$
t_4	$\parallel$	$\parallel$	$\parallel$	$\parallel$

$m_1 : t_1 \mapsto t_2$

d_{12}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\parallel$	$\parallel$	$\rightarrow$
t_2	$\parallel$	$\parallel$	$\parallel$	$\parallel$
t_3	$\parallel$	$\parallel$	$\parallel$	$\parallel$
t_4	$\leftarrow$	$\parallel$	$\parallel$	$\parallel$

$m_1 : t_1 \mapsto t_4$

The assumptions of the hypotheses are shown below the tables. In this case, d_{11} is obtained by assuming m_1 to be sent from t_1 to t_2, while d_{12} assumes m_1 is 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 w.r.t. 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. E.g., 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:

d_{21}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow$	$\parallel$	$\rightarrow$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\parallel$
t_3	$\parallel$	$\parallel$	$\parallel$	$\parallel$
t_4	$\leftarrow$	$\parallel$	$\parallel$	$\parallel$

$m_1 : t_1 \mapsto t_2$; $m_2 : t_1 \mapsto t_4$

d_{22}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow$	$\parallel$	$\parallel$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_3	$\parallel$	$\parallel$	$\parallel$	$\parallel$
t_4	$\parallel$	$\leftarrow$	$\parallel$	$\parallel$

$m_1 : t_1 \mapsto t_2$; $m_2 : t_2 \mapsto t_4$

d_{23}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\parallel$	$\parallel$	$\rightarrow$
t_2	$\parallel$	$\parallel$	$\parallel$	$\rightarrow$
t_3	$\parallel$	$\parallel$	$\parallel$	$\parallel$
t_4	$\leftarrow$	$\leftarrow$	$\parallel$	$\parallel$

$m_1 : t_1 \mapsto t_4$; $m_2 : t_2 \mapsto t_4$

After period 1, we update D_{cur} with $\{d_{21}, d_{22}, d_{23}\}$. Post-processing operations are performed at the end of each period to remove all the assumptions, test conditional dependencies, and delete redundant hypotheses.

The algorithm then proceeds to period 2. After period 2 and the post-processing, D_{cur} contains 5 hypotheses. After period 3, these 5 hypotheses remain in D_{cur}:

d_{81}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow^?$	$\rightarrow^?$	$\rightarrow$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\parallel$
t_3	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_4	$\leftarrow$	$\parallel$	$\leftarrow^?$	$\parallel$

d_{82}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\parallel$	$\rightarrow^?$	$\rightarrow$
t_2	$\parallel$	$\parallel$	$\parallel$	$\rightarrow$
t_3	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_4	$\leftarrow$	$\leftarrow^?$	$\leftarrow^?$	$\parallel$

d_{83}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow^?$	$\parallel$	$\rightarrow$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_3	$\parallel$	$\parallel$	$\parallel$	$\rightarrow$
t_4	$\leftarrow$	$\leftarrow^?$	$\leftarrow^?$	$\parallel$

d_{84}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow^?$	$\rightarrow^?$	$\rightarrow$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_3	$\leftarrow$	$\parallel$	$\parallel$	$\parallel$
t_4	$\leftarrow$	$\leftarrow^?$	$\parallel$	$\parallel$

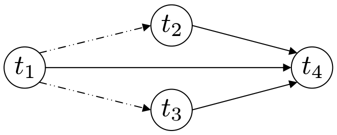

Figure 4. Dependency graph of the simple model

d_{85}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow^?$	$\rightarrow^?$	$\parallel$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_3	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_4	$\parallel$	$\leftarrow^?$	$\leftarrow^?$	$\parallel$

Each of these hypotheses is the most specific ones that satisfy all the instances. However, because of the limited number of instances, the algorithm cannot converge. Optionally, we may consider their least upper bound d_{LUB} (which no longer guarantees optimality) as the result:

d_{LUB}	t_1	t_2	t_3	t_4
t_1	$\parallel$	$\rightarrow^?$	$\rightarrow^?$	$\rightarrow$
t_2	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_3	$\leftarrow$	$\parallel$	$\parallel$	$\rightarrow$
t_4	$\leftarrow$	$\leftarrow^?$	$\leftarrow^?$	$\parallel$

d_{LUB} satisfies $\forall s \in T, t \in T . d_{LUB}(s,t) = d_{81}(s,t) \sqcup d_{82}(s,t) \sqcup \cdots \sqcup d_{85}(s,t)$, 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 rounds and 700 event-pair executions of tasks and messages. An example of event-pair for task A from a trace is as follows:

```
A] Begin execution for round 27 at time 798.015
A] End execution for round 27 at time 798.017
```

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 extrapolated into a dependency graph (Figure 5). We used this dependency graph to prove properties (i.e. dependencies and operation mode of tasks, such as conjunction or disjunction) of the system. The output of the algorithm confirmed some properties that were known in advance; e.g. Tasks A and B are

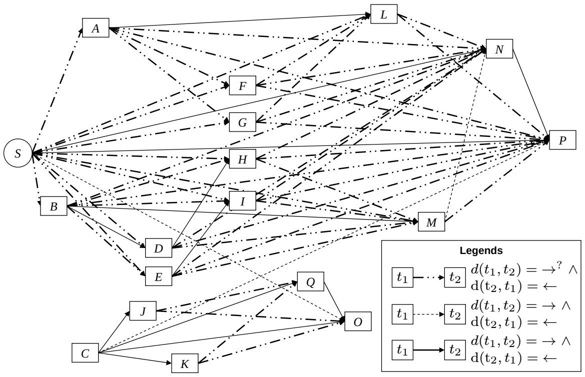
A
L
N
F
G
S
H
P
I
B
M
D
E
Q
J
O
C
K
Legends
t_1
t_2
$d(t_1,t_2) = \rightarrow^? \wedge$
$\mathrm{d}(t_2,t_1) = \leftarrow$
t_1
t_2
$d(t_1,t_2) = \rightarrow \wedge$
$\mathrm{d}(t_2,t_1) = \leftarrow$
t_1
t_2
$d(t_1,t_2) = \rightarrow \wedge$
$\mathrm{d}(t_2,t_1) = \leftarrow$

Figure 5. Dependency graph of a GM model

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 brake is pressed, then seat belts must tighten," are known a priori at design time. However, some properties, for example the data dependency between Q and O (refer to Figure 5) come from the integrations of a CAN bus or an OSEK scheduler, are not known at design time.

The dependency relations that we obtained 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 reduces the pessimisms 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. Note that since the system is a black box, the correctness of the generated model is based on how exhaustive the traces are.

The run time of our implementation is tested in Windows XP with a Pentium M 1.7 GHz processor and 1 GB memory with different bounds. When unbounded, the exponential algorithm runs for minutes without stopping.

Bound	Run time (sec)	Bound	Run time (sec)
1	0.220	64	5.899
4	0.471	100	12.608
16	1.202	120	16.294
32	2.573	150	19.048

4. Properties of the Algorithm

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

Proof. This can be proved by transforming the general Boolean Satisfiability Problem (SAT), known to be *NP*-complete, to the learning problem.

To show the transformation, we need to prove the following fact first:

Boolean expression $x_1 \vee x_2 \vee \cdots \vee x_j$ is satisfiable if and only if $(v \vee x_1 \vee x_2 \vee \cdots \vee x_i) \wedge (\neg v \vee x_{i+1} \vee \cdots \vee x_j)$ ($0 < i < j$, v is a fresh boolean variable) is satisfiable. The proof is simple because $x \vee y$ is satisfiable if and only if $(v \vee x) \wedge (\neg v \vee y)$ is satisfiable, and x and y themselves can be boolean expressions.

For each clause in the SAT problem, we separate variables and their negations (if any) into two clauses with the above mechanism, introducing fresh variables when necessary. This results in clauses with only positive literals or negative literals. From now on, we only consider boolean expressions of this form. Let n be the total number of variables including the introduced ones, and m be the total number of clauses.

We now transform any given SAT problem with n variables $\{x_1, x_2, \cdots, x_n\}$ and m clauses into a trace learning problem with $n+1$ tasks $\{t_1, t_2, \cdots, t_n, \tau\}$. This transformation is conducted by examining every clause in the boolean expression, and creating a corresponding period in the trace. For the i-th clause, let $\{x_{i1}, x_{i2}, \cdots, x_{ik}\}$ be the k variables in it. If they all appear positive, then in the period, $t_{i1}, t_{i2}, \cdots, t_{ik}$ are created (order irrelevant), followed by a single message m_i, followed by τ. This means m_i was sent by one of $t_{i1}, t_{i2}, \cdots, t_{ik}$ to τ, therefore reflecting the "or" relation between them. If $\{x_{i1}, x_{i2}, \cdots, x_{ik}\}$ appear negative instead, the creation of this period is similar: τ comes first, followed by m_i, followed by $t_{i1}, t_{i2}, \cdots, t_{ik}$. This implies that m_i was sent by τ to one of $t_{i1}, t_{i2}, \cdots, t_{ik}$. This process is repeated for every clause in the boolean expression, yielding m total periods in the trace. The learning problem seeks to find the most specific hypotheses that match all the periods, reflecting the "and" relation between clauses.

With any algorithm to learn the dependency from the above-constructed trace, we can either give an assignment to all the variables so that the boolean expression is satisfied or decide that the expression is unsatisfiable:

- If the following is true for any of the most specific hypotheses in the result, then the boolean expression is satisfiable:

  $$\forall t_i \in \{t_1, t_2, \cdots, t_n\}. d(t_i, \tau) \sqsubset_V \leftrightarrow^?$$

  An assignment to make the expression true is as follows:

  $$x_i = \begin{cases} true & d(t_i,\tau) = \rightarrow \ \vee \ d(t_i,\tau) = \rightarrow^? \\ false & d(t_i,\tau) = \leftarrow \ \vee \ d(t_i,\tau) = \leftarrow^? \\ any & d(t_i,\tau) = \parallel \end{cases}$$

  This assignment is due to the observation that if t_i (sometimes) determines τ but it never depends on τ, then x_i should be assigned true; if the opposite holds, then x_i should be assigned false.

- If the following is true for all of the resulting most specific hypotheses, then the original boolean expression is unsatisfiable:

  $$\exists t_i \in \{t_1, t_2, \cdots, t_n\}.d(t_i, \tau) = \leftrightarrow^{?}$$

  This is because any assignment to satisfy the boolean expression would require x_i to be assigned both true and false, which is impossible.

SAT is *NP*-complete, and the transformation from SAT to the learning problem is polynomial, so the learning problem is *NP*-hard. □

Theorem 2 (Correctness). The algorithm (with or without heuristics) guarantees correctness. I.e., if I is the set of instances in the trace, and D^* is the set of hypotheses that the algorithm returns, then $\forall d^* \in D^*.M(d^*, I)$.

Proof. This can be proved by induction on the number of steps (i.e., periods) in the algorithm. The induction claim is that in every step, the hypotheses match all the instances up to that step.

- Base case: The claim is true initially when we have learned no period, and the initial set of hypothesis is simply $\{d_\perp\}$.
- Induction step: Assuming that up to the previous period the claim is true, then in the current period, any hypothesis that does not match the dependencies in this period will be generalized. So at the end of this period, the remaining hypotheses match this period as well as all previous periods. □

Theorem 3 (Optimality and completeness). The algorithm without heuristics guarantees optimality and completeness. I.e., if I is the set of instances in the trace, and D^* is the set of hypotheses that the algorithm returns, then $\forall d \in D.M(d, I) \Rightarrow (\exists d^* \in D^*.d^* \sqsubseteq_D d)$.

Proof. This can be proved by induction on the number of steps (i.e., periods). Let the first period be $i_1 \in I$, the second period be $i_2 \in I$, etc. Let I_k be the set of first k periods $\{i_1, i_2, \cdots, i_k\} \subseteq I$. Let D_k^* be the set of hypotheses learned from all the elements of I_k.

- Base case: As the result of learning from the first period, D_1^* is obtained. By construction, the algorithm constrains D_1^* so that the hypotheses in it are the most specific that match $I_1 = \{i_1\}$. For each message in i_1, the algorithm explores all possible sender-receiver pairs as the hypotheses' assumptions. Therefore, $\forall d \in D$, if $M(d, I_1)$, then d matches all those messages with a specific combination of sender-receiver pairs. This combination must have been explored by the above exhaustive learning algorithm. Hence, $\exists d_1^* \in D_1^*.d_1^* \sqsubseteq_D d$.
- Induction step: Assume that after learning period k, $\forall d \in D.M(d, I_k) \Rightarrow (\exists d_k^* \in D_k^*.d_k^* \sqsubseteq_D d)$. We now show that the claim is also true for $k+1$. For any $d \in D$, if $M(d, I_{k+1})$, then $M(d, I_k) \wedge M(d, \{i_{k+1}\})$. Because of the induction assumption and $M(d, I_k)$, $\exists d_k^* \in D_k^*.d_k^* \sqsubseteq_D d$. For period $k+1$, the algorithm generalizes d_k^* only as much as necessary to match i_{k+1} (refer to the optimality requirement in Section 3.1). Hence, $\exists d_{k+1}^* \in D_{k+1}^*.d_{k+1}^* \sqsubseteq_D d$ (d_{k+1}^* is a generalized from d_k^* with instance i_{k+1}). □

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

Proof. This can be proved by induction on the number of generalizations (i.e. messages). We do not consider the post-processing after every period because it only modifies the assumptions but not the hypotheses.

- Base case: Initially, the algorithm starts with $\{d_\perp\}$ no matter what the bound is, so the claim is trivially true because $d_\perp = \bigsqcup\{d_\perp\}$.
- Induction step: Assume that after learning message m_i, D_i is obtained with the bound set to b, and d_i is the only hypothesis obtained with the bound set to 1. According to the induction assumption, $d_i = \bigsqcup D_i$.

  When examining message m_{i+1} (whether it is in the same period as m_i or not), the algorithm with the bound set to b generalizes all hypotheses in D_i as much as necessary so that they match m_{i+1}. This results in D_{i+1}. On the other hand, with the bound set to 1, the algorithm returns d_{i+1}. We now prove that ($d_{i+1} \sqsubseteq_D \bigsqcup D_{i+1}$) and ($\bigsqcup D_{i+1} \sqsubseteq_D d_{i+1}$).

  1. All hypotheses in D_{i+1} match the messages up to m_{i+1}, so $\bigsqcup D_{i+1}$ also matches all messages up to m_{i+1}. d_{i+1} is the most specific hypothesis that matches all messages up to m_{i+1} by construction, so $d_{i+1} \sqsubseteq_D \bigsqcup D_{i+1}$.
  2. According to the induction assumption, $d_i = \bigsqcup D_i$, so $\forall d \in D_i.d \sqsubseteq_D d_i \sqsubseteq_D d_{i+1}$. For any $d' \in D_{i+1}$, because the algorithm generalize d' only as much as necessary from a d in D_i, and both d' and d_{i+1} match all messages up to m_{i+1}, then $d' \sqsubseteq_D d_{i+1}$. Because this is true for any $d' \in D_{i+1}$, $\bigsqcup D_{i+1} \sqsubseteq_D d_{i+1}$.

  Because of 1 and 2, $d_{i+1} = \bigsqcup D_{i+1}$ holds after the algorithm processes m_{i+1}. Therefore, $d^* = \bigsqcup D^*$ after the algorithm processes all the messages in the trace. □

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 $d_1^* = d_2^*$.

Proof. As a result of the above lemma, if the algorithm returns a set with a single hypothesis $\{d_1^*\}$, then $d_2^* = \bigsqcup\{d_1^*\} = d_1^*$. □

Due to the convergence theorem, if we knew that the trace allows the algorithm to converge with a unique, most-specific dependency function, we only need to set the algorithm's bound of hypotheses to 1. If, because of the scheduler's properties that are not intended in the original design, or because of insufficiency of the trace, the algorithm cannot converge, then the bound affects the result's optimality.

The algorithm without heuristics is exponential in both the number of tasks and the largest number of messages in a period. We also show that no polynomial algorithm exists to compute the optimal solution by proving this problem to be *NP*-hard. Heuristics is then necessary to make the algorithm feasible. With heuristics, we give the complexity of the algorithm without proof, which is $O(mb^2 + mbt^2)$, where m is the number of messages in the whole 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 without heuristics is correct and optimal. With additional heuristics, the algorithm converges in polynomial time (in the number of messages and the bound).

Though the motivation behind this paper originated from an automotive application, the algorithm could be extended to other applications as well. This algorithm could also be extended to version space techniques if negative instances were provided in the execution traces.

References

[1] OSEK OS version 2.2.3 specification, 2006.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[3] R. Bosch. CAN specification version 2.0, 1991.

[4] J. G. Huselius, J. Andersson, H. Hansson, and S. Punnekkat. Automatic generation and validation of models of legacy software. In *12:th IEEE International Conference RTCSA*, Sydney, Australia, August 2006.

[5] S. Kanajan, H. Zeng, C. Pinello, and A. Sangiovanni-Vincentelli. Exploring trade-off's between centralized versus decentralized automotive architectures using a virtual integration environment. In *DATE'06*, March 2006.

[6] T. A. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *International Conference on Knowledge Capture (K-CAP)*, pages 36–43, Banff, Alberta, Canada, 2003.

[7] E. A. Lee and A. L. Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), 1998.

[8] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[9] P. Pop. *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems*. PhD thesis, Dept. of Computer and Information Science, Linköping University, Sweden, 2003.

[10] R. L. Rivest and R. E. Schapire. Diversity-based inference of finite automata. *J. ACM*, 41(3):555–589, 1994.

[11] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.