Automatic Abstraction for Verification of Cyber-Physical Systems

Robert A. Thacker, Kevin R. Jones, Chris J. Myers
University of Utah
{thacker,kjones,myers}@vlsigroup.ece.utah.edu

Hao Zheng
University of South Florida
zheng@cse.usf.edu

ABSTRACT

Models of cyber-physical systems are inherently complex since they must represent hardware, software, and the physical environment. Formal verification of these models is often precluded by state explosion. Fortunately, many important properties may only depend upon a relatively small portion of the system being accurately modeled. This paper presents an automatic abstraction methodology that simplifies the model accordingly. Preliminary results on a fault-tolerant temperature sensor are encouraging.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Software/Program Verification

1. INTRODUCTION

Verification of cyber-physical systems is complicated by their heterogeneous nature as well as their sheer complexity. Cyber-physical systems include hardware, software, and a physical environment, so a formal model must integrate all of these concerns. Unfortunately, modeling a system with all of its details results in state explosion. Therefore, it is necessary to automatically abstract the model to include only those details necessary to verify the property of interest. Constructing such a model is the focus of this paper.

One candidate model for cyber-physical systems are *hybrid automata* [3, 4], but their use of invariants make them cumbersome to generate from higher level descriptions. *Hybrid Petri nets* are another alternative [5], but their use of separate continuous places and transitions makes them also difficult to generate. The *labeled hybrid Petri net* (LHPN) model has been developed and applied to the verification of analog/mixed-signal circuits, and compilers have been developed from VHDL-AMS as well as SPICE simulation data [8, 9, 14]. This model includes both Boolean variables for representing digital circuits and continuous variables for representing analog circuits. In [12], the LHPN model is extended to support discrete variables for representing software variables as well as expressions to check and modify them. These extensions allow for both hardware and software to be represented in a single model along with their continuous physical environment. A compiler and a model checker have been developed to support the use of this model for the verification of cyber-physical systems [12, 13].

Verifying LHPN models of cyber-physical systems at full detail is not possible due to the state explosion problem. This paper presents an automatic abstraction technique for simplifying these LHPN models. The basic idea is to apply *LHPN transformations* to remove details from the model that are irrelevant to the property of interest. These transformations are inspired by transformations for ordinary Petri nets [11] and timed Petri nets [15]. They are also inspired from various static analysis techniques used by compilers [1]. Other related work includes reduction techniques for timed and hybrid automata described in [6, 7, 10].

This paper is organized as follows. Section 2 describes a motivating example of a fault-tolerant temperature sensor for a nuclear reactor cooling system. Section 3 introduces the LHPN model. Section 4 briefly introduces our state space exploration algorithm, and Section 5 briefly describes a compiler to generate LHPN models. The core of the paper is Section 6 which presents our LHPN transformations. Section 7 presents results for our example, and Section 8 presents our conclusions and future goals.

2. MOTIVATING EXAMPLE

A traditional cyber-physical system example is the cooling system for a nuclear reactor [2]. In this example, the temperature of the nuclear reactor core is monitored, and when the temperature is too high, one of two control rods is inserted to cool the reactor core. In our modified version of the example, there are two temperature sensors to add fault tolerance. Namely, each temperature sensor is periodically sampled and if at any point the temperature difference between them is too large, it is assumed that one of them has become faulty and the reactor is shut down. A block diagram for this example is shown in Fig. 1.

This example is interesting because it includes an interface with a physical environment (i.e., the temperature sensors), mixed-signal components (i.e., the analog/digital converters (ADCs)), digital components (i.e., the microcontroller), and embedded software (i.e., the program running on the microcontroller). The verification problem is to determine if a temperature mismatch error can occur even when the temperature sensors are operating correctly. On the surface,

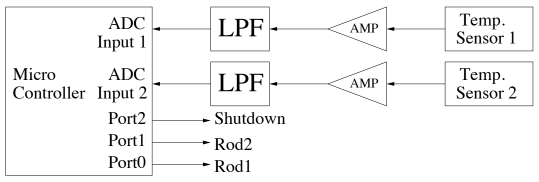


Figure 1: Fault-tolerant temperature sensor for a nuclear reactor.

this does not appear to be a problem, but there are a number of implementation details that make this not so obvious. First, there is typically only one ADC on a microcontroller which is multiplexed to sample from each ADC input one at a time. This means that the temperature sensors are not sampled at exactly the same time. A second problem is that since the comparison of the results is not done with a single atomic instruction at the assembly level, it is possible that the results are not even from the same sampling cycle.

3. THE LHPN MODEL

An LHPN is a Petri net model originally developed to represent *analog/mixed-signal* (AMS) circuits [8, 14], and it has recently been extended to represent software [12, 13]. The LHPN model is inspired by both hybrid Petri nets [5] and hybrid automata [3]. While the models in this paper could certainly be represented using these traditional formalisms, we have found it extremely difficult to develop an automatic compiler that targets these formalisms. Therefore, the LHPN model has been developed with the goal of being easy to generate from various higher-level representations of cyber-physical systems. An LHPN is a tuple $N = \langle P, T, T_f, B, X, V, \Delta, \dot{V}, F, L, M_0, S_0, Y_0, Q_0, R_0\rangle$:

- P : is a finite set of places;
- T : is a finite set of transitions;
- $T_f \subseteq T$: is a finite set of failure transitions;
- B : is a finite set of Boolean variables;
- X : is a finite set of discrete integer variables;
- V : is a finite set of continuous variables;
- Δ : is a finite set of rate variables;
- $\dot{V} : V \to \Delta$ is the mapping of variables to their rates;
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- L : is a tuple of labels defined below;
- $M_0 \subseteq P$ is the set of initially marked places;
- $S_0 : B \to \{0, 1, \bot\}$ is the initial value of each Boolean;
- $Y_0 : X \to (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{\infty\})$ is the initial range of values for each discrete variable;
- $Q_0 : V \to (\mathbb{Q} \cup \{-\infty\}) \times (\mathbb{Q} \cup \{\infty\})$ is the initial range of values for each continuous variable;
- $R_0 : \Delta \to (\mathbb{Q} \cup \{-\infty\}) \times (\mathbb{Q} \cup \{\infty\})$ is the initial range of rates of change for each continuous variable.

A simple LHPN is shown in Fig. 2. The places are labeled $p0$, $p1$, and $p2$ with $p0$ and $p2$ initially marked. The transitions are labeled $t0$, $t1$, and $t2$ with transition $t2$ being a failure transition. The flow relation is represented by the

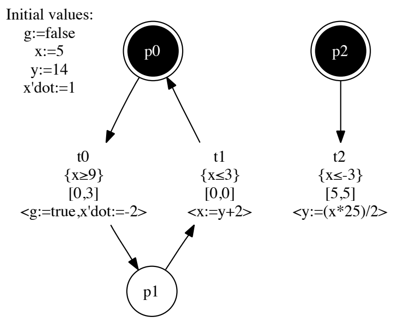


Figure 2: Example LHPN.

arcs connecting the places and the transitions. This example has a Boolean variable, g, which is initially **false**, a discrete variable, y, with an initial value of 14, and a continuous variable, x, with an initial value of 5 and rate of 1.

Every transition $t \in T$ has a *preset* denoted by $\bullet t = \{p \mid (p, t) \in F\}$ and a *postset* denoted by $t\bullet = \{p \mid (t, p) \in F\}$. Presets and postsets for places are defined similarly. The functions $\bullet\mathcal{T} = \bigcup_{t\in\mathcal{T}} \bullet\, t$ and $\mathcal{T}\bullet = \bigcup_{t\in\mathcal{T}} t\bullet$ apply to sets of transitions. The set of all possible successor transitions reachable from a set of transitions $\mathcal{T}$ is defined with the recursive function $post(\mathcal{T}) = (\mathcal{T}\bullet\bullet) \cup (post(\mathcal{T}\bullet\bullet))$. Similarly, $pre(\mathcal{T}) = (\bullet\bullet\mathcal{T}) \cup (pre(\bullet\bullet\mathcal{T}))$ defines the set of all possible predecessor transitions from which $\mathcal{T}$ may be reached. Transitions in an LHPN that are connected by the flow relation are said to reside in the same *process*, and all transitions in the same process as transition t can be determined using the recursive function $proc(\{t\}) = pre(\mathcal{T}) \cup post(\mathcal{T}) \cup proc(pre(\mathcal{T}) \cup post(\mathcal{T}))$. The LHPN in Fig. 2 includes two processes (i.e., $\{t0, t1\}$ and $\{t2\}$).

Before defining the labels formally, let us first introduce the grammar used by these labels. The numerical portion of the grammar is defined as follows:

$$\begin{aligned}\chi \ ::= \ & c_i \mid \infty \mid x_i \mid v_i \mid \dot{v}_i \mid (\chi) \mid -\chi \mid \chi + \chi \mid \\ & \chi - \chi \mid \chi * \chi \mid \chi/\chi \mid \chi\hat{}\chi \mid \chi\%\chi \mid \\ & \mathrm{NOT}(\chi) \mid \mathrm{OR}(\chi, \chi) \mid \mathrm{AND}(\chi, \chi) \mid \\ & \mathrm{XOR}(\chi, \chi) \mid \mathrm{INT}(\phi)\end{aligned}$$

where c_i is a rational constant from $\mathbb{Q}$, x_i is a discrete variable, and v_i is a continuous variable. The function $\dot{v}_i$ returns the rate variable associated with the continuous variable v_i. The functions NOT, OR, AND, and XOR are bit-wise logical operations, and they are only applicable to integers and assume a 2's complement format with arbitrary precision. The function INT converts a Boolean **true** value to an integer 1 and **false** value to an integer 0. The set $\mathcal{P}_\chi$ is defined to be all formulas that can be constructed from the χ grammar.

The Boolean part of the grammar is as follows:

$$\begin{aligned}\phi \ ::= \ & \textbf{true} \mid \textbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \chi = \chi \mid \\ & \chi \geq \chi \mid \chi > \chi \mid \chi \leq \chi \mid \chi < \chi \mid \mathrm{BIT}(\chi, \chi)\end{aligned}$$

where b_i is a Boolean variable, and $\mathrm{BIT}(\alpha_1, \alpha_2)$ extracts bit α_2 from α_1. The set $\mathcal{P}_\phi$ is defined to be all formulas that can be constructed from the ϕ grammar.

The analysis algorithm restricts enabling conditions to a subset of the χ and ϕ grammars. The χ_e grammar does not allow continuous variables to be used, nor does it allow Boolean expressions to be converted into integers. The set $\mathcal{P}_{\chi_e}$ is defined to be all formulas that can be constructed from the χ_e grammar. The ϕ_e grammar is restricted such that enabling conditions only allow continuous variables to appear on the left side of relations of the form $v_i \geq \chi_e$ or $v_i \leq \chi_e$. This guarantees that the right side of these relations remains constant between transition firings as time advances. The set $\mathcal{P}_{\phi_e}$ is defined to be all formulas that can be constructed from the ϕ_e grammar.

Each transition in an LHPN is labeled with an enabling condition as well as a set of assignments. These are formally defined using the tuple $L = \langle En, D, BA, XA, VA, RA \rangle$:

- $En : T \to \mathcal{P}_{\phi_e}$ labels each transition $t \in T$ with an enabling condition.
- $D : T \to \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{\infty\})$ labels each transition $t \in T$ with a lower and upper delay bound, $[d_l(t), d_u(t)]$.
- $BA : T \times B \to \mathcal{P}_\phi$ labels each transition $t \in T$ and Boolean variable $b \in B$ with the Boolean assignment made to b when t fires.
- $XA : T \times X \to \mathcal{P}_\chi$ labels each transition $t \in T$ and discrete variable $x \in X$ with the discrete variable assignment that is made to x when t fires.
- $VA : T \times V \to \mathcal{P}_\chi$ labels each transition $t \in T$ and continuous variable $v \in V$ with the continuous variable assignment that is made to v when t fires.
- $RA : T \times \Delta \to \mathcal{P}_\chi$ labels each transition $t \in T$ and continuous rate variable $\dot{v} \in \Delta$ with the rate assignment that is made to $\dot{v}$ when t fires.

For convenience, $AV = B \cup X \cup V \cup \Delta$ denotes the set of all variables, and $AA = BA \cup XA \cup VA \cup RA$ denotes all assignments to these variables. Note that most assignments are vacuous (i.e., reassign the existing value) and are therefore not represented in the graphical representation. This is defined formally as follows: $vac(t, v) \Leftrightarrow (AA(t, v) = v)$.

Transition $t0$ from the first process of Fig. 2 has an enabling condition of $\{x \geq 9\}$. The delay of this transition varies from 0 to 3 time units. When $t0$ fires, the rate, $\dot{x}$, is assigned to -2. The firing of transition $t0$ also assigns the Boolean variable g to **true**. The firing of transition $t1$ assigns the continuous variable x to the value of the expression $y + 2$. The firing of $t2$ results in a discrete variable assignment to y which sets its value to the value of the expression $(x * 25)/2$. Note that this assignment scales a continuous variable and assigns a truncated value to an integer.

The formal semantics are in [12, 13], and briefly described here using the example in Fig. 2. In the initial state, $p0$ and $p2$ are marked; g is **false**; y has a value of 14; x has a value of 5 and changing at a rate of 1. Since $t0$ is guarded by the Boolean expression $\{x \geq 9\}$ and $t2$ by $\{x \leq -3\}$, and neither is satisfied in the initial state, no transitions are initially enabled. After 4 time units, x reaches a value of 9, and $\{x \geq 9\}$ becomes true, enabling transition $t0$. Since the delay assignment on $t0$ is [0,3], transition $t0$ fires within the next 3 time units. When $t0$ fires, g is set to **true**, x is set to change at a rate of -2, and the marking is moved from $p0$ to $p1$. In this new state, $p1$ is marked, but transition $t1$ is not yet enabled. Since x now has a value between 9 and 12, it can take anywhere from 3 to 4.5 time units for $\{x \leq 3\}$ to become true. When this happens, transition $t1$ fires instantly setting x to a value of 15. The right process is monitoring if the value of x ever drops below -3 for five time units or more. If this happens then transition $t2$ fires indicating a failure. While this does not occur for these parameters, it fails if the delay of transition $t1$ is set to 8.

4. STATE SPACE EXPLORATION

State space exploration is required to analyze and verify properties of LHPNs. This exploration is complicated by the fact that LHPNs typically have an infinite number of states. Therefore, to perform state space exploration on LHPNs, this infinite number of states must be represented by a finite number of convex state equivalence classes called *state sets*. In particular, these state sets use *zones* represented using *difference bound matrices* (DBMs). State exploration proceeds as a depth first search of the state space, and it terminates either when all state sets have been found or when a failure transition is fired. It should be noted, however, that state space exploration for LHPNs is undecidable, so it actually may not terminate. Details about the state exploration method have been reported earlier in [8, 12].

5. LHPN COMPILER

Constructing LHPN models by hand is quite tedious. To address this issue, we have developed a compiler that maps a higher-level description into the low-level LHPN model. An LHPN model for a cyber-physical system is composed of three parts: the hardware (both analog and digital), the software, and the physical environment. The hardware and physical environment models tend to be more stable and more reusable. For example, a model of a microcontroller can be reused in every system that uses that microcontroller. On the other hand, the software is typically unique for every system and may be updated and modified easily and thus often. Software is developed in some high-level language that is then compiled to an assembly language for the microprocessor or microcontroller on which it runs. Precise analysis of timing is often critical for the correctness of cyber-physical systems. This information is not available in the high-level language used for software development, so our analysis must focus at the assembly language level where timing of individual instructions is known. However, compilers for high-level languages are leveraged to produce the assembly language input to our compiler.

Our compiler takes as input a language definition that defines how each instruction is mapped to LHPN constructs and a description of the process using this language. The language definition only needs to be constructed once for each type of process. Using this definition, the compiler directly translates each instruction into a portion of an LHPN which is then stitched together to form a model of the complete process. Constructs have been developed for representing subroutines, interrupts, and threads using this modeling approach, and details can be found in [13].

Modeling the temperature sensor requires three processes. The first models the environment, the second the processor ADC hardware, and the last the software. To simplify the presentation, only the portion of the model related to the temperature sensors is presented. The environment model is shown in Fig. 3(a). This uses the `set_rate` instruction that has operands of an enabling condition, continuous variable, new rate for the variable, and lower and upper delay on the rate change, and it creates a transition with these parameters. Neglecting the control rods, the reactor temperature is simply modeled as a triangle wave. The temperature is allowed to fall at a rate of two temperature units per time unit until reaching a value of 2200. The temperature then rises at a rate of two temperature units per time unit until reaching a value of 9800. At this point, the temperature begins to decrease again. The analog circuitry in the model (the low pass filters and amplifiers) are encapsulated in this model, and the variable *temp* is provided as the input to the ADC subsystem. The compiled LHPN process for the environment model is shown in Fig. 3(b). The description of the ADC subsystem is omitted due to space limitations [13]. The LHPN process for part of the ADC subsystem is shown in Fig. 3(c). In particular, this model shows two of the ten operating modes, as the others are similar. Finally, the assembly software and corresponding LHPN process is shown in Fig. 4. This model implements the initialization and redundant temperature sensor check, but does not implement the cooling rod control loop. Storing 48 to the ADCTL register initiates a sample of an0 through an3. The program then busy-waits until it receives the adc_ccf flag from the ADC subsystem, which shows up as the high order bit of a read from the ADCTL register. Once a complete cycle has finished, the program then repetitively reads the contents of ADR1 and ADR2 and compares their values. If they are within a tolerance, the loop repeats. If not, an error code is written to PORTB and the program enters a stall loop. Note that `;@ fail_set` marks the following transition, t36, as a failure transition.

6. LHPN TRANSFORMATIONS

Most system models are too complicated to analyze at a low-level of detail, but too high a level of detail may not allow the properties of interest to be verified. For example, verification of programs in a high-level language such as C omit timing information, but verification of assembly level software quickly results in state explosion. This section presents several LHPN transformations that can simplify the system model by removing unnecessary details. The LHPN transformations essentially transform assembly language programs back into higher level expressions while preserving the timing of critical operations. While most LHPN transformations presented do not change the state space, in the worst-case, they are conservative and do not produce false positive results. This section illustrates the LHPN transformations using the LHPN model for the fault-tolerant temperature sensor shown in Figs. 3 and 4.

6.1 Preliminaries

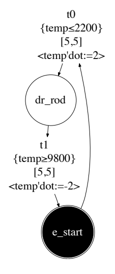
A transition t is said to *read* a variable v if it contains any reference to v other than its own vacuous assignment. This is defined formally as follows:

$$reads(t, v) \Leftrightarrow (v \in sup(En(t))\vee$$
$$\exists v' \in AV.(\neg vac(t, v') \wedge v \in sup(AA(t, v')))).$$

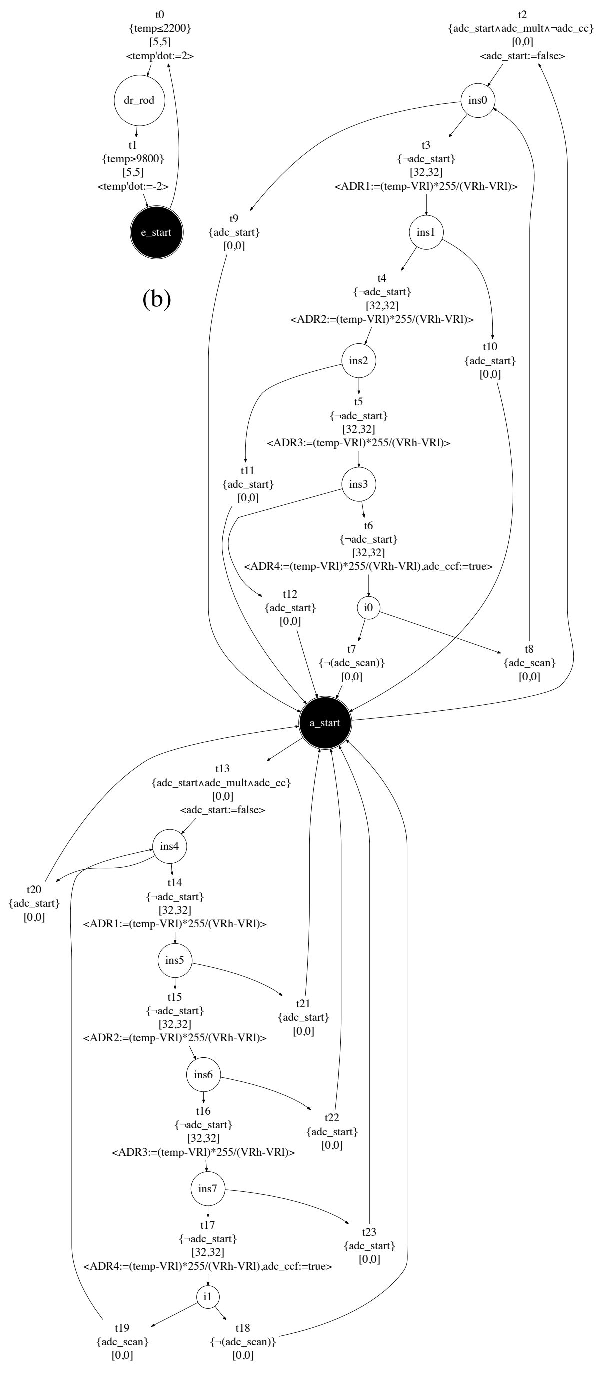
Note that the function $sup(e)$ returns the set of all variables that occur in the expression e.

Many of the LHPN transformations can only be applied to variables that are local to a process. Formally, a variable

```
include <example.inst>
e_start set_rate  temp<=2200 temp 2 5 5
dr_rod  set_rate  temp>=9800 temp -2 5 5
        link      e_start
```
(a)



(b)



(c)

Figure 3: (a) Reactor environment model. LHPN representing the fault-tolerant temperature sensor (b) environment and (c) ADC circuitry.

```
;@ include <6811.inst>
main ldab    #48
      stab    ADCTL
test ldab    ADCTL
      bpl     test
loop ldab    ADR1
      ldaa    ADR2
     sba
     adda    #7
     cmpa    #14
     bls     loop
;@   fail_set
     ldab    #7
     stab    PORTB
term bra     term
```

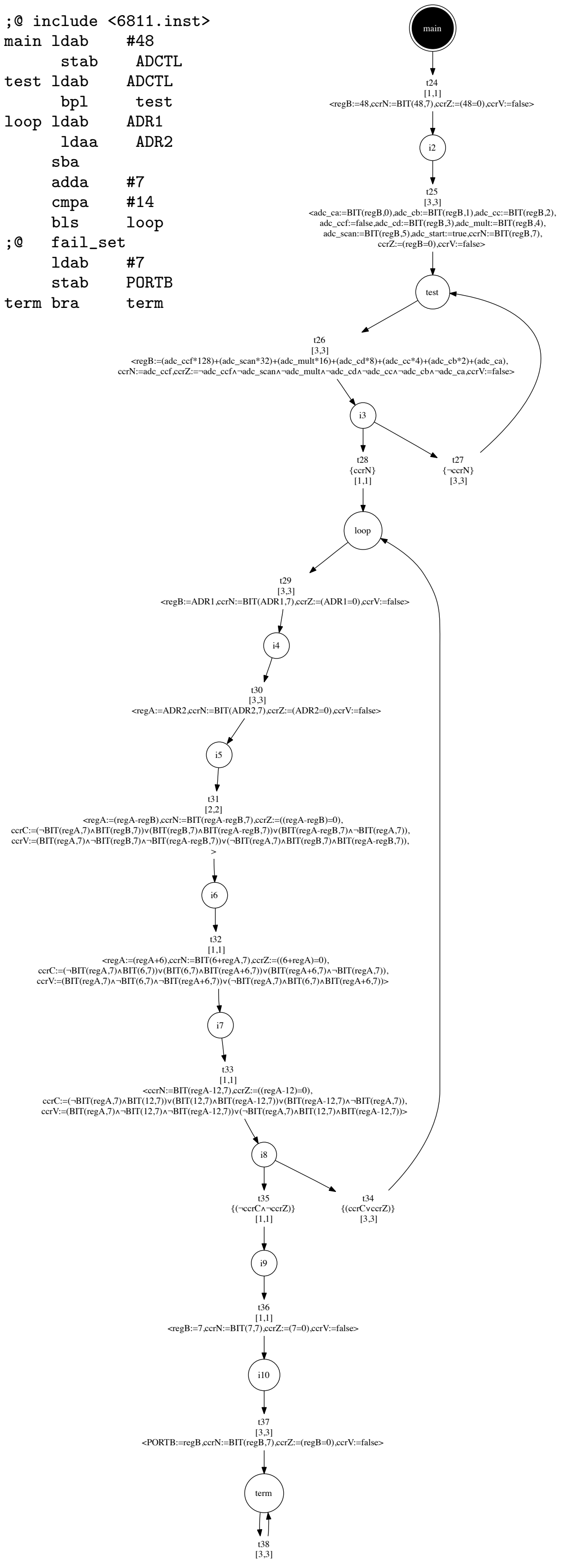

main
t24
[1,1]
<regB:=48,ccrN:=BIT(48,7),ccrZ:=(48=0),ccrV:=false>
i2
t25
[3,3]
<adc_ca:=BIT(regB,0),adc_cb:=BIT(regB,1),adc_cc:=BIT(regB,2),
adc_ccf:=false,adc_cd:=BIT(regB,3),adc_mult:=BIT(regB,4),
adc_scan:=BIT(regB,5),adc_start:=true,ccrN:=BIT(regB,7),
ccrZ:=(regB=0),ccrV:=false>
test
t26
[3,3]
<regB:=(adc_ccf*128)+(adc_scan*32)+(adc_mult*16)+(adc_cd*8)+(adc_cc*4)+(adc_cb*2)+(adc_ca),
ccrN:=adc_ccf,ccrZ:=¬adc_ccf∧¬adc_scan∧¬adc_mult∧¬adc_cd∧¬adc_cc∧¬adc_cb∧¬adc_ca,ccrV:=false>
i3
t28
{ccrN}
[1,1]
t27
{¬ccrN}
[3,3]
loop
t29
[3,3]
<regB:=ADR1,ccrN:=BIT(ADR1,7),ccrZ:=(ADR1=0),ccrV:=false>
i4
t30
[3,3]
<regA:=ADR2,ccrN:=BIT(ADR2,7),ccrZ:=(ADR2=0),ccrV:=false>
i5
t31
[2,2]
<regA:=(regA-regB),ccrN:=BIT(regA-regB,7),ccrZ:=((regA-regB)=0),
ccrC:=(¬BIT(regA,7)∧BIT(regB,7))∨(BIT(regB,7)∧BIT(regA-regB,7))∨(BIT(regA-regB,7)∧¬BIT(regA,7)),
ccrV:=(BIT(regA,7)∧¬BIT(regB,7)∧¬BIT(regA-regB,7))∨(¬BIT(regA,7)∧BIT(regB,7)∧BIT(regA-regB,7)),
>
i6
t32
[1,1]
<regA:=(regA+6),ccrN:=BIT(6+regA,7),ccrZ:=((6+regA)=0),
ccrC:=(¬BIT(regA,7)∧BIT(6,7))∨(BIT(6,7)∧BIT(regA+6,7))∨(BIT(regA+6,7)∧¬BIT(regA,7)),
ccrV:=(BIT(regA,7)∧¬BIT(6,7)∧¬BIT(regA+6,7))∨(¬BIT(regA,7)∧BIT(6,7)∧BIT(regA+6,7))>
i7
t33
[1,1]
<ccrN:=BIT(regA-12,7),ccrZ:=((regA-12)=0),
ccrC:=(¬BIT(regA,7)∧BIT(12,7))∨(BIT(12,7)∧BIT(regA-12,7))∨(BIT(regA-12,7)∧¬BIT(regA,7)),
ccrV:=(BIT(regA,7)∧¬BIT(12,7)∧¬BIT(regA-12,7))∨(¬BIT(regA,7)∧BIT(12,7)∧BIT(regA-12,7))>
i8
t35
{(¬ccrC∧¬ccrZ)}
[1,1]
t34
{(ccrC∨ccrZ)}
[3,3]
i9
t36
[1,1]
<regB:=7,ccrN:=BIT(7,7),ccrZ:=(7=0),ccrV:=false>
i10
t37
[3,3]
<PORTB:=regB,ccrN:=BIT(regB,7),ccrZ:=(regB=0),ccrV:=false>
term
t38
[3,3]

Figure 4: Assembly code and LHPN process for the fault-tolerant temperature sensor software.

v is local with respect to the process containing transition t as defined below:

$$local(t,v) \Leftrightarrow (v \in (B \cup X) \wedge \\ \forall t' \in (T - proc(\{t\})).(\neg reads(t',v) \wedge vac(t',v))).$$

Intuitively, this means the variable is neither referenced nor assigned in any other process.

Some transformations can only be applied to variables that are locally written within a process. References to these variables within that process can be reshuffled, but the timing of assignments must be maintained. Formally, a variable v is locally written with respect to the process containing transition t as defined below:

$$lw(t,v) \Leftrightarrow (v \in (B \cup X) \wedge \\ \forall t' \in (T - proc(\{t\})).vac(t',v)).$$

The function $LW(t) = \{v \in AV \mid lw(t,v)\}$ returns the set of all variables which are locally written with respect to the process containing transition t.

As an artifact of compilation and LHPN transformations, expressions are often constructed that can be simplified. The function *simplify*(e) performs basic arithmetic and logical simplifications when all operands are constant or in some cases when one operand is a constant (i.e., 0 and 1 for arithmetic or **true** and **false** when logical).

While applying LHPN transformations, it is occasionally necessary to substitute an expression for a variable. The function $replace(e,v,e')$ substitutes the expression e' for every occurrence of the variable v in the expression e. It then applies the function *simplify*(e) to the resulting expression. The function $replace(t,v,e)$ performs $replace(En(t),v,e)$ and $replace(AA(t,v'),v,e)$ for all v' in AV.

A sequence of transitions $\rho = (t_0, t_1, ..., t_n)$ is defined to be a *path* if $\forall i \in \{0,1,...,n\}.((t_i \in T) \wedge ((i=n) \vee (t_{i+1} \in t_i \bullet\bullet))$. The set of paths $\Pi(N)$ is the set of all paths ρ defined by the flow relation within an LHPN. Note that this is *not* an execution sequence, but a graphically connected ordered set of transitions.

The LHPN transformations presented in this paper are assumed to be applied only to LHPNs in which each process may have choice but not concurrency (i.e., $\forall t \in T.|t\bullet| = |\bullet t| = 1$). This assumption is reasonable since all LHPNs generated by compilation satisfy this property. Concurrency is achieved by the use of communicating processes.

6.2 Remove Write Before Write

When a variable is written and then rewritten before it is referenced, the original calculation is unnecessary, and the assignment can be made vacuous without changing the observable behavior. This is formally defined as follows:

Transformation 1. ***(remove write before write)*** *Consider a transition t and a variable v. If*

1. $\neg vac(t,v)$,
2. $local(t,v)$,
3. $\neg\exists(t_0,t_1,...,t_n) \in \Pi(N).(t_0 = t) \wedge reads(t_n,v) \wedge \forall i \in \{1,2,...,n-1\}.vac(t_i,v)$

then $AA(t,v) := v$.

This transformation is illustrated with the LHPN fragment in Fig. 5. If transition t performs a non-vacuous assignment to a variable v that is local with respect to the

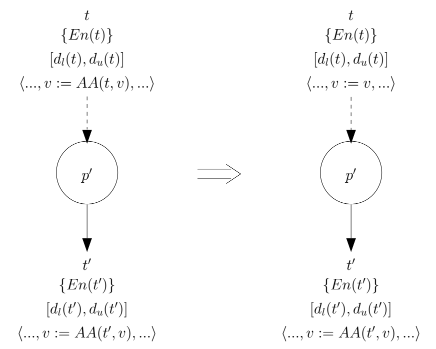


Figure 5: Remove write before write.

process associated with t, and along all paths that start with t and end with a transition that reads variable v, a non-vacuous assignment occurs to v before the read, then the assignment to v on transition t can be made vacuous.

Consider the first two transitions of the software model shown in Fig. 6(a). These represent the instructions `ldab #48` and `stab ADCTL`. The three condition codes ccrN, ccrZ, and ccrV are written in both transitions, but not referenced during the firing of $t25$. This is quite common as condition code registers are set by most instructions, but they are read primarily by branch instructions. Applying Transformation 1, these assignments are made vacuous. Since these variables are also overwritten by transition $t26$, they can also be made vacuous on transition $t25$. Therefore, this portion of the LHPN is simplified as shown in Fig. 6(b) after applying this transformation to the entire software model.

6.3 Local Assignment Propagation

The exact timing of local variable assignments is unimportant unless they affect the calculation of global variables or depend on global variables. It is, therefore, possible to push variable assignments forward to perform "just in time" assignment. This requires that all transitions immediately preceding the target transition make exactly the same assignment to the variable, and that none make changes to the support set of the assignment expression. This is defined formally as follows:

Transformation 2. ***(local assignment propagation)*** *Consider an assignment $v := AA(t, v)$ on a transition t. If*

1. $\neg vac(t, v)$,
2. $local(t, v)$,
3. $sup(AA(t, v)) \subseteq LW(t)$,
4. $\forall t'' \in \bullet(t\bullet).AA(t'', v) = AA(t, v)$, *and*
5. $\forall t'' \in \bullet(t\bullet).\forall v' \in (sup(AA(t, v)) - \{v\}).vac(t'', v')$

then

1. $\forall t' \in (t \bullet \bullet).replace(t', v, AA(t, v))$, *and*
2. $\forall t'' \in \bullet(t\bullet).AA(t'', v) := v$.

This transformation is illustrated with the LHPN fragment in Fig. 7. If $v := AA(t, v)$ is a non-vacuous assignment to a variable that is local with respect to the process associated with transition t, all variables v' that are in the

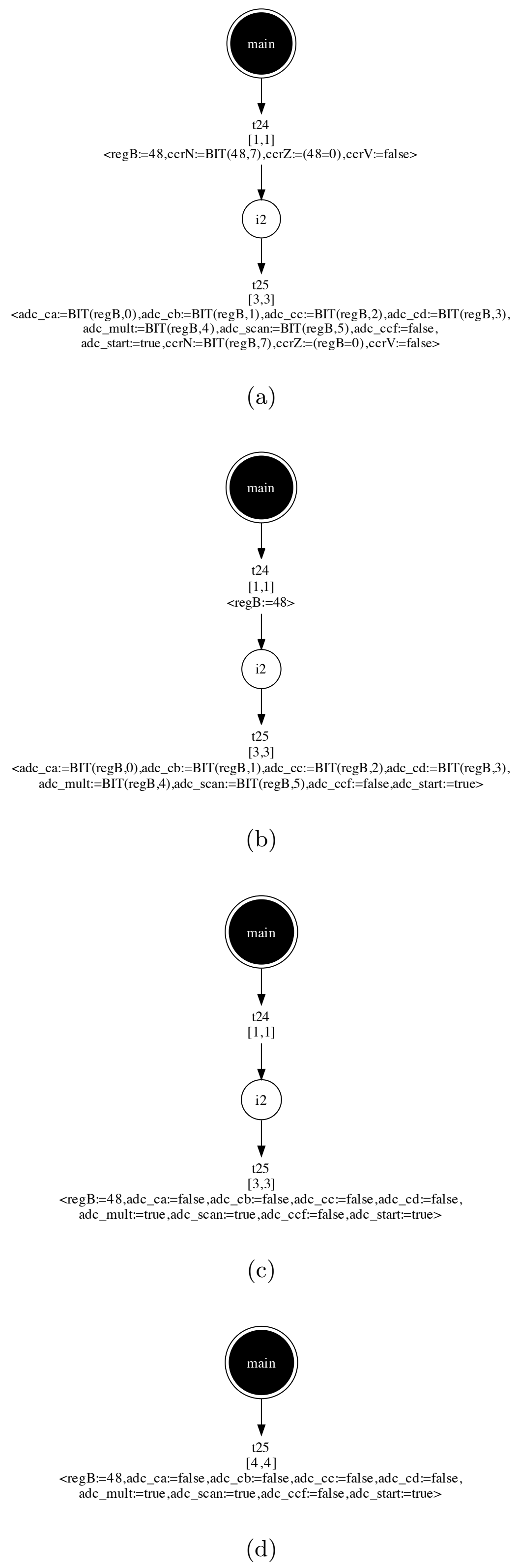

main
t24
[1,1]
<regB:=48,ccrN:=BIT(48,7),ccrZ:=(48=0),ccrV:=false>
i2
t25
[3,3]
<adc_ca:=BIT(regB,0),adc_cb:=BIT(regB,1),adc_cc:=BIT(regB,2),adc_cd:=BIT(regB,3),
adc_mult:=BIT(regB,4),adc_scan:=BIT(regB,5),adc_ccf:=false,
adc_start:=true,ccrN:=BIT(regB,7),ccrZ:=(regB=0),ccrV:=false>
(a)
main
t24
[1,1]
<regB:=48>
i2
t25
[3,3]
<adc_ca:=BIT(regB,0),adc_cb:=BIT(regB,1),adc_cc:=BIT(regB,2),adc_cd:=BIT(regB,3),
adc_mult:=BIT(regB,4),adc_scan:=BIT(regB,5),adc_ccf:=false,adc_start:=true>
(b)
main
t24
[1,1]
i2
t25
[3,3]
<regB:=48,adc_ca:=false,adc_cb:=false,adc_cc:=false,adc_cd:=false,
adc_mult:=true,adc_scan:=true,adc_ccf:=false,adc_start:=true>
(c)
main
t25
[4,4]
<regB:=48,adc_ca:=false,adc_cb:=false,adc_cc:=false,adc_cd:=false,
adc_mult:=true,adc_scan:=true,adc_ccf:=false,adc_start:=true>
(d)

Figure 6: Software initialization transitions. (a) Initial model, (b) after applying Transformation 1, (c) after applying Transformation 2, and (d) after applying Transformation 3.

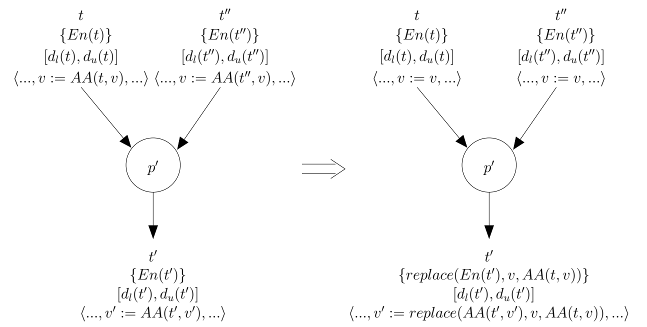


Figure 7: Local assignment propagation example.

support of this expression are locally written, all transitions in $\bullet(t\bullet)$ make the same assignment to v, and all variables other than v in the support of $AA(t, v)$ are not assigned in these transitions, then all occurrences of v in transition t' can be replaced with $AA(t, v)$ and the assignment to v on all transitions in $\bullet(t\bullet)$ can be made vacuous.

Consider the application of Transformation 2 to transition $t24$ shown in Fig. 6(b). The variable regB is local with respect to the software process, and the assignment to regB on transition $t24$ references no variables that are written outside this process. Therefore, this assignment can be propagated forward, and its value pushed into the expressions in $t25$. The result after simplifying the expressions is shown in Fig. 6(c).

6.4 Remove Vacuous Transitions

After the previous two transformations, it is often the case that transitions no longer include any non-vacuous assignments. Therefore, these transitions only mark the passage of time. When this occurs, the delay can be pushed into the following transitions, and the transition and its following place are collapsed. For this transformation to occur, all enabling conditions of the transition and all of its successors transitions must only involve locally written variables. This prevents the enabling conditions from becoming disabled once they are enabled. This is defined formally as follows:

TRANSFORMATION 3. ***(remove vacuous transitions)*** *Consider a transition t. If*

1. $\forall v \in AV.vac(t, v)$,
2. $(\bullet t)\bullet = \bullet(t\bullet) = \{t\}$,
3. $sup(En(t)) \subseteq LW(t)$,
4. $\forall t_i \in (t \bullet \bullet).sup(En(t_i)) \subseteq LW(t)$, *and*
5. $t \notin T_F$.

then

1. $T = T - \{t\}$
2. $P = P - t\bullet$
3. $\forall t_i \in (t \bullet \bullet).d_l(t_i) = d_l(t) + d_l(t_i)$
4. $\forall t_i \in (t \bullet \bullet).d_u(t_i) = d_u(t) + d_u(t_i)$
5. $\forall t_i \in (t \bullet \bullet).En(t_i) = En(t) \wedge En(t_i)$
6. $F = (F - R1) \cup R2$ *where* $R1 = \{(p, t) \in F \mid p \in \bullet t\} \cup \{(t, p') \in F \mid p' \in t\bullet\} \cup \{(p', t_i) \mid (p' \in t\bullet) \wedge (t_i \in t \bullet\bullet)\}$ *and* $R2 = \{(p, t_i) \mid (p \in \bullet t) \wedge (t_i \in t \bullet \bullet)\}$

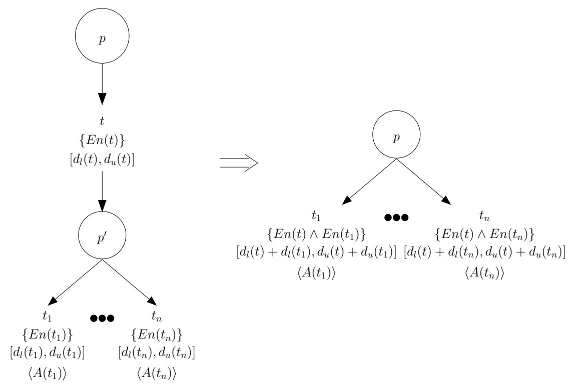


Figure 8: Remove vacuous transitions.

This transformation is illustrated with the LHPN fragment in Fig. 8. If transition t includes only vacuous assignments, the structure of the net is exactly as shown, the support of the enabling condition of t and its successor transitions include only locally written variables, and t is not a failure transition then t can be removed and its enabling condition and delay can be pushed forward.

As shown in Fig. 6(c), after applying the previous two transformations, transition $t24$ has only vacuous assignments. Therefore, it can be removed using Transformation 3. The result is shown in Fig. 6(d).

6.5 Correlated Variables

Occasionally, two or more variables are closely correlated. Every time one of them is assigned, the other is assigned to a value which is easily derived from the other. That value may be the same or a clear function of the other. In either case, if they are always assigned at the same time and have the same relationship to each other *every time* that they are assigned, it is not necessary to maintain both variables. This is defined formally as follows:

TRANSFORMATION 4. ***(correlated variables)*** *Consider the variables v and v' from B or X. If*

$$\forall t \in T.(AA(t, v') = f(AA(t, v)) \vee (vac(t, v) \wedge vac(t, v')))$$

where $f(x)$ is any clearly defined function of one variable, including the identity function, then $\forall t \in T.replace(t, v', f(v))$.

Let us consider the LHPN fragment shown in Fig. 6(d). This is the only write to the ADCTL register, so other than the handshaking signals adc_start and adc_ccf, the rest of the ADC control bits are only set here. This makes it clear that they are highly correlated and that adc_ca, adc_cb, adc_cc, adc_cd, adc_mult, and adc_scan can be reduced to a single variable. If Transformation 4 is applied pairwise to these variables, it is discovered that adc_ca, adc_cb, and adc_cd can be replaced with adc_cc, and adc_mult and adc_scan are replaced with ¬adc_cc. Consider transitions $t2$ and $t13$, shown in Fig. 9(a). This transformation results in the changes shown in Fig. 9(b).

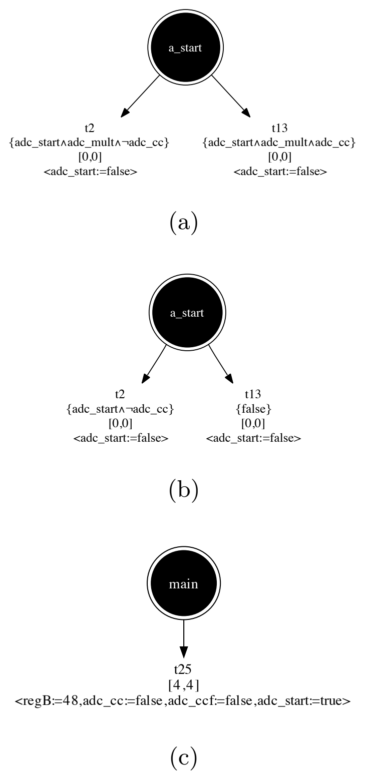


Figure 9: ADC enabling transition (a) before and (b) after Transformation 4. (c) Changes to the initialization transition after Transformation 5.

6.6 Remove Unread Variables

When a variable is written but never read, the assignment can be made vacuous with no change in observable behavior. This is formally defined as follows:

Transformation 5. ***(remove unread variables)*** *Consider variable v from B or X. If $\forall t \in T.\neg reads(t, v)$ then $AV = AV - \{v\}$.*

After Transformation 4, adc_ca, adc_cb, adc_cd, adc_mult, and adc_scan are no longer read, and their assignments can be removed as shown in Fig. 9(c).

6.7 Remove Dead Transitions and Places

A transition is *dead* if it can never fire. For example, if the enabling condition of a transition is a constant false, this transition is dead. Similarly, if there exist no tokens in any predecessor places to a transition, it is also dead as its preset can never become marked. Dead transitions can be safely removed from the LHPN. This is defined formally as follows:

Transformation 6. ***(remove dead transitions)*** *Consider a transition t. If $(En(t) = false) \vee (pre(\bullet t) \cap M_0) = \emptyset$ then*

1. *$T = T - \{t\}$*
2. *$F = F - \{(p, t) \in F \mid p \in \bullet t\} \cup \{(t, p) \in F \mid p \in t\bullet\}$.*

Removing dead transitions often results in places that no longer have any transitions in their postset. These places can be removed. This is formally defined as follows:

Transformation 7. ***(remove dead places)*** *Consider a place p. If $(p\bullet) = \emptyset$, the net can be transformed as follows:*

1. *$P = P - \{p\}$*
2. *$F = F - \{(t, p) \in F \mid t \in \bullet p\}$.*

In the example shown in Fig. 9(b), after applying Transformation 4 transition $t13$ can no longer fire and is removed. After removing transition $t13$ from the LHPN shown in Fig. 3(b), transitions $t14$ through $t23$ can no longer have their preset place become marked, so they are dead as well and can be removed. This leaves places ins4 through ins7 and i1 dangling, so they can be removed as well. This reduces the size of the LHPN for the ADC subsystem in half.

6.8 Remove Arc After Failure Transitions

Since the firing of a failure transition ends state space exploration, any transitions following it are unnecessary. Therefore, it is safe to remove the edge between a failure transition and the place in its postset as follows:

Transformation 8. ***(remove arc after failure transitions)*** *Consider a transition $t \in T_f$. The net can be transformed as follows: $F = F - \{(t, p) \in F \mid p \in t\bullet\}$.*

Since $t36$ is a failure transition, the arc after this transition can be removed resulting in the removal of all the following transitions and places using Transformations 6 and 7.

6.9 Timing Bound Normalization

As explained in Section 4, our state space exploration finds states sets rather than individual states. Representing irregular sets of states can be difficult. Therefore, it is advantageous to have timing bounds that encapsulate a range of behaviors. This can be accomplished using a timing bound normalization in which the delay assignments are enlarged such that the bounds are a multiple of a given normalization factor k. This, however, is an abstraction since it introduces new behavior into the reachable state sets. However, it is conservative in that no false positive verification results occur. This transformation is formally defined as follows:

Transformation 9. ***(timing bound normalization)*** *For a normalizing factor k, adjust the delay assignment for each transition t as follows:*

1. *$d_l(t) = \lfloor d_l(t)/k \rfloor * k$*
2. *$d_u(t) = \lceil d_u(t)/k \rceil * k$*

For our example, a normalization with $k = 5$ performs well. This value is chosen as it is the delay bound on the environment transitions and is smaller than most of the other delays in the LHPN model.

6.10 Other LHPN Transformations

There are several other LHPN transformations that have been developed but not presented in detail here due to space limitations. These include:

- Constant enabling condition determination,
- Removing dominated transitions,
- Removing vacuous loops, and
- Removing unnecessary transitions with global variables in their enabling conditions.

For details about these transformations, please see [13].

The final reduced LHPN model of the ADC subsystem is shown in Fig. 10, and the final reduced LHPN model of the software process is shown in Fig. 11. Note that the environment model is unchanged.

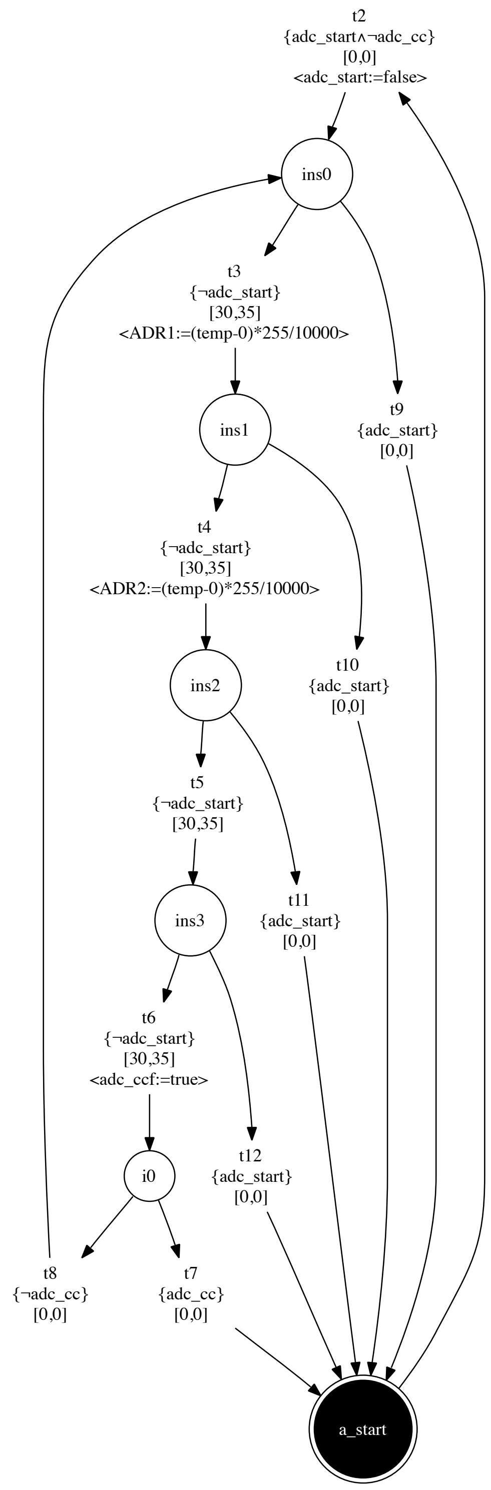


Figure 10: Final reduced LHPN model for the ADC process.

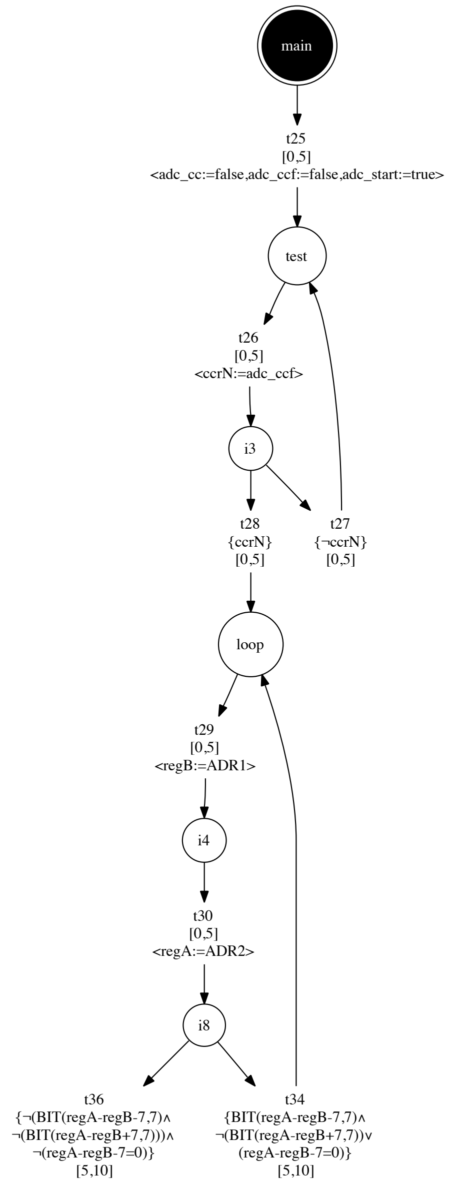


Figure 11: Final reduced LHPN model for the software process.

7. RESULTS

We have updated the LEMA verification tool to support automatic compilation, transformation, and verification of models of cyber-physical systems. As a case study, we have applied LEMA to the fault-tolerant temperature sensor with several variations in parameter values. The results are shown in Table 1. For each case, the number of state sets found, runtime in seconds (including compilation, transformation, and verification), and whether it verifies to be correct are reported. Recall that the property being verified is that the reactor never shuts down since the temperature sensors are assumed to be perfect in the LHPN model.

Table 1: Verification results for the reactor example.

Parameters	State sets	Time (s)	Verifies
Original LHPN	>1e6	>12 hrs	?
Reduced LHPN	35563	311	Yes
W/o init. loop	5	0.52	No
9-bit ADCs	945	0.79	No
Slow ADC	38	0.38	No
temp rates $[-4, 4]$	32	0.41	No
temp rates $[-4, 4]$, 7-bit ADCs	21787	94.5	Yes

The original LHPN could not be verified after 12 hours and finding more than 1 millions state sets. The reduced LHPN model with normalized parameters completes in 311 seconds (about 5 minutes) after finding 35563 state sets, and it verifies to be correct. A naive designer might initiate the ADC conversion and immediately launch into the main software loop. `LEMA` takes 0.52 seconds and finds 5 state sets in determining that this design fails. The reason for the failure is that ADR1 and ADR2 are sampled before they can be loaded from the ADC, so regA and regB are loaded with the uninitialized reset values. Suppose a new microcontroller is substituted into a mature design which increases the ADC resolution from 8-bits to 9-bits. In this case, if the tolerance value of ± 7 is not increased to reflect the greater resolution, the system fails. `LEMA` requires 0.79 seconds and found 945 state sets to discover this flaw. Another possible hardware change might be a microcontroller with a slower ADC system. Suppose instead of taking 32 clock cycles to make a conversion it requires 64 cycles. `LEMA` encounters 38 state sets in 0.38 seconds to find this error. New experimental data may determine that the rate of change of the temperature is ± 4 instead of ± 2, as in the existing environment model, the cumulative error between readings exceeds the allowed ± 7, and the system fails. `LEMA` takes 0.41 seconds and finds 32 state sets to find this failure. As a final variation, consider an attempt to rectify the higher temperature slew rate by employing a lower resolution ADC. This combination proves successful requiring 94.5 seconds and 21787 state sets to verify. Overall these results indicate that the correctness of this fault-tolerant temperature sensor is quite sensitive to parameter choices.

8. CONCLUSION

This paper presents a methodology for automatically abstracting models of cyber-physical systems. The models are described using a user-defined language inspired by assembly code. They are automatically compiled into a LHPN model that is capable of representing hardware, software, and the environment in a single formalism. The model complexity is reduced using LHPN transformations that in most cases do not change the result of verification, and in the worst-case do not generate any false positive results. This methodology is applied to a practical case study of a fault-tolerant temperature sensor. While preliminary results are encouraging, there are still a number of interesting directions for future research. In particular, there are numerous additional LHPN transformations that can be developed. Also, abstraction refinement should be automated. Finally, more case studies on a variety of model types should be investigated.

9. ACKNOWLEDGEMENTS

This research is supported by NSF Grants No. CNS-0930225 and CNS-0930510, SRC contract 2008-TJ-1851, and an SRC Graduate Fellowship.

10. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Tech. and Tools.* Addison-Wesley, 1988.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Comp. Sci.*, 138(1):3–34, 1995.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer, 1992.

[4] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proc. of the IEEE*, 91(1):11 – 28, Jan 2003.

[5] R. David and H. Alla. On hybrid Petri nets. *Discrete Event Dynamic Systems: Theory and Applications*, 11(1–2):9–40, Jan. 2001.

[6] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. RTSS'96*, pages 73–81. IEEE Computer Society Press, 1996.

[7] J.-C. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. *Sci. Comp. Prog.*, 47(2-3):203–220, 2003.

[8] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda. Verification of analog/mixed-signal circuits using labeled hybrid Petri nets. In *Proc. International Conference on Computer Aided Design (ICCAD)*, pages 275–282. IEEE Computer Society Press, 2006.

[9] S. Little, D. Walter, and C. Myers. Analog/mixed-signal circuit verification using models generated from simulation traces. In *Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *LNCS*, pages 114–128. Springer, 2007.

[10] H. Maka, G. Frehse, and B. H. Krogh. Polyhedral domains and widening for verification of numerical programs. In *NSV-II: Second International Workshop on Numerical Software Verification*, 2009.

[11] T. Murata. Petri nets: Properties, analysis, and applications. In *Proc. of the IEEE*, volume 77, pages 541–580, 1989.

[12] R. Thacker, C. Myers, K. Jones, and S. Little. A new verification method for embedded systems. In *Proc. International Conference on Computer Design (ICCD)*. IEEE Computer Society Press, 2009.

[13] R. A. Thacker. *A New Verification Method for Embedded Systems.* PhD thesis, U. of Utah, Jan. 2010.

[14] D. Walter, S. Little, C. Myers, N. Seegmiller, and T. Yoneda. Verification of analog/mixed-signal circuits using symbolic methods. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 27(12):2223–2235, 2008.

[15] H. Zheng, E. Mercer, and C. J. Myers. Modular verification of timed circuits using automatic abstraction. 22(9):1138–1153, Sept. 2003.