Applied Verification: The Ptolemy Approach

Chihhong Patrick Cheng
Teale Fristoe
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-41
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-41.html

April 19, 2008

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0647591 (CSR-SGER), and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

Applied Verification: The Ptolemy Approach

Chihhong Patrick Cheng
EECS, UC Berkeley
EE, National Taiwan Univ.
patrickj@eecs.berkeley.edu

Teale Fristoe
EECS, UC Berkeley
fristoe@eecs.berkeley.edu

Edward A. Lee
EECS, UC Berkeley
eal@eecs.berkeley.edu

April 19, 2008

Abstract

This paper addresses the difficulty that designers of embedded software systems face when doing formal verification. Existing theories and practices in verification are powerful, but when applying formal techniques, the use of detailed mathematical model descriptions in verification greatly increase the burden on system designers; construction of such models may be time consuming and error prone. We lay the groundwork for solving this problem by providing a mapping from actor models to mathematical models suitable for verification; the conversion is automatic with minimal human intervention. Meanwhile, the interactions between the verification model and its environment can guide us in designing how the implementation model interprets the raw data from sensors and to actuators, allowing us to reuse the verification model as the basis of its implementation model. Following these strategies, the productivity of designers and the correctness of designs can be maintained simultaneously.

1 Introduction

Designing embedded systems according to the model-based approach [16, 35], detailed implementations are synthesized from models. If the model and the synthesis technique are correct, then the implementation is guaranteed to be correct. In this paper, we address the issue of ensuring that a model is correct.

Verification [33] is the process of determining whether a design conforms to its specification; techniques may include simulation and testing. However, for many embedded systems, violating specifications can be lethal, expensive, or both, so the above two techniques may not provide the necessary confidence. Formal approaches help solve this problem by providing rigorous mathematical means to explore system behavior exhaustively, offering stronger claims regarding the correctness of the system.

Theoretically, modeling and verification should be tightly bound. In practice, an irreconcilable tension exists between modeling convenience and formal verification, and tools tend towards one extreme or the other.

Tools emphasizing modeling convenience focus on the productivity of designers, time-to-market constraints, and design re-use through modeling expressiveness, readability, extensibility, and synthesizability. However, their lack of verification can be problematic for safety critical systems.

Tools supporting formal verification emphasize precision, tractability, and compositionality at the expense of ease of construction. Furthermore, verification techniques often only handle control flow, when data transformation is equally important for model correctness, so redundant modeling becomes necessary.

Our approach, *applied verification*, incorporates the benefits of both standard approaches: designers can easily model systems and then rigorously verify them. Two components make this new approach a reality:

- To maintain ease of design while still allowing the use of formal verification techniques, we have developed an automatic mapping from commonly used higher level components to mathematical models used for verification, which requires little human intervention (sections 3, 4). This is especially important because many designers are domain experts with little understanding of formal verification techniques. The converted models can be fed to model checkers which will perform formal verification to test the correctness of control flow. Behavior which cannot be checked with model checkers, like data correctness, can be tested using existing simulation mechanisms.
- Our *design for verification* methodology shows how to design a model that can be used for verification and then as the basis for implementation. When verifying a model, we need to characterize the behavior of the environment; we then mimic these characteristics in the implementation's sensors and actuators, so we can use the same model for both purposes, allowing maximal component reuse (section 5).

We use Ptolemy II [14] as the framework for applied verification (section 6) and provide a case study (section 7). Ptolemy II is an open-source software package for modeling and simulating concurrent, real-time, embedded systems. It facilitates the design process with a component assembly framework and a graphical user interface. Its provided actor library, which is easily extendible, allows designers to quickly construct models. We believe that Ptolemy II can serve as the front-end design tool for any verification engine, given a conversion mechanism between Ptolemy II models and the necessary mathematical models. Our work allows model checking on some models in the *synchronous reactive* (SR) domain and the *discrete event* (DE) domain to compose *finite state machines* (FSMs) and *modal models*. We leverage the code generation infrastructure of Ptolemy II to integrate the NuSMV [9] model checker for the verification of synchronous and real-time systems (section 6.2). Our provided instructions show how other model checkers can be easily integrated into Ptolemy II.

2 Preliminaries

We use the following notation.

- Given a set V, the size (number of elements) of V is denoted as $|V|$, and $\mathbf{2}^V$ is the powerset of V.
- The symbol ϕ represents the empty set.
- Mathematical notation '$\exists$**1**' represents "*exists only one element*".
- The `verbatim` font specifies textual strings.
- Square brackets represent the atomic proposition, e.g., $[a == 3]$.
- 'iff' is a shorthand for "if and only if".
- **0** and **1** are indistinguishable from the boolean values `false` and `true`, respectively.
- $\bigwedge$ is the symbol for Cartetian product.

2.1 Ptolemy II Models, Actors and Domains

Systems in Ptolemy II are built using components called *actors*. Though models often feature no text, the following *abstract syntax* of an actor will facilitate our discussion.

Definition 1 *An actor is a tuple* $A = (E, P, Para, R)$.

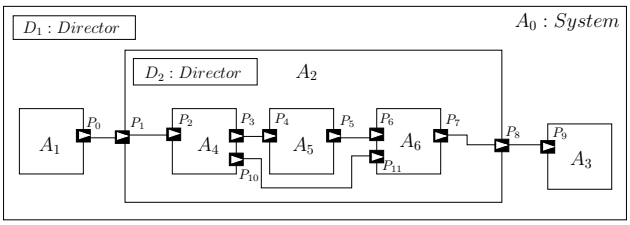


Figure 1: A system featuring actors and directors.

- E *is the set of (inner) actors.*
- P *is the set of* ports*.*
- $Para = \{(name, attribute)\}$ *is the set of* parameters *specifying properties.*
- $R = \{(p_i, Para_{ij}, p_j) | p_i \in \hat{P}, p_j \in \dot{P}, \hat{A} = \{\hat{E}, \hat{P}, \hat{Para}, \hat{R}\} \in E, \dot{A} = \{\dot{E}, \dot{P}, \dot{Para}, \dot{R}\} \in E\} \cup \{(p_i, Para_{ij}, p_j) | p_j \in \hat{P}, p_i \in P, \hat{A} = \{\hat{E}, \hat{P}, \hat{Para}, \hat{R}\} \in E\} \cup \{(p_i, Para_{ij}, p_j) | p_i \in \hat{P}, p_j \in P, \hat{A} = \{\hat{E}, \hat{P}, \hat{Para}, \hat{R}\} \in E\}$ *are the* channels *between (1) two inner actors' ports or (2) a port of the actor and a port of the actor's inner actor. For each channel* $(p_i, Para_{ij}, p_j)$, $Para_{ij} = \{(name, attribute)\}$ *is the set of parameters for the channel.*

Ptolemy II uses *directors* to support heterogeneous modeling. Directors control the execution order of actors and mediate their communication. A director is an actor $A_{dir} = (\phi, \phi, Para_{dir}, \phi)$, where $Para_{dir}$ contains an element $(\texttt{director}, semantics)$. The $semantics$ attribute may be an element of the set $\{\texttt{FSM}, \texttt{SR}, \texttt{DE}, \texttt{CT}, \texttt{PTIDES}\}$ [1]. Heterogeneous models consist of different directors at different levels. Actors are *domain polymorphic* in that their semantics depend on the director controlling them.

A *system* is merely a composite actor; its hierarchical structure is defined by the recursive definition of its inner actor set. In fig. 1, the system is represented as $A_0 = \{E_0, P_0, Para_0, R_0\}$ where $E_0 = \{A_1, A_2, A_3, D_1\}$, $P_0 = \phi$, $R_0 = \{(P_0, P_1), (P_8, P_9)\}$. A_1 is an atomic actor, while A_2 is composite.

2.2 Models Commonly Used in Verification

Generally, four types of models are used for formal verification: graph models, game models, stochastic models, and timed models [22]. We focus on graph models and timed models (graph models with continuous variables), and introduce *Kripke structures* and *communicating timed automata* (CTA), which are commonly used by existing model checkers.

Definition 2 *A Kripke structure over a set of atomic propositions* $AP = \{p_1, \ldots, p_n\}$ *is a tuple* $M = (S, S_0, R, L)$.

- S *is a finite set of* states*.*
- $S_0 \subseteq S$ *is the set of* initial states*.*
- $R \subseteq S \times S$ *is the set of* transition relations*.*
- $L = S \rightarrow 2^{AP}$ *is the* labeling function *that associates* $s \in S$ *to the set of atomic propositions* $AP' \subseteq AP$ *that are true in* s*.*

Definition 3 *A system of communicating timed automata is a tuple* $S = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$*, where* $\mathcal{A}_i = \{Q_i, C_i, Sync_i, q_i, Jump_i, Inv_i\}$ *is an automaton with the following constraints.*

[1] The set listed includes common directors, but Ptolemy II features more directors to support a rich set of MoCs.

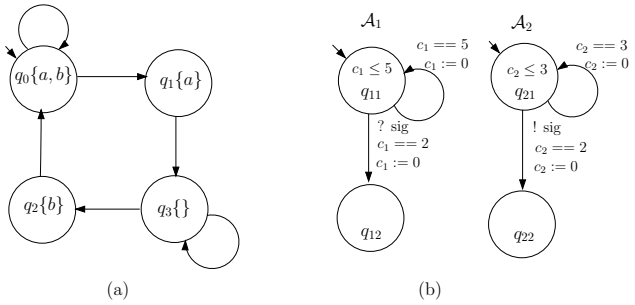


Figure 2: Examples of Kripke structure (a) and communicating timed automata (b).

- Q_i *is a finite set of* modes *(*locations*).*
- $C_i = \{c_{i_1}, \ldots, c_{i_m}\}$ *is the set of* clock variables*.*
- $Sync_i = \{s_{i_1}, \ldots, s_{i_n}\}$ *is the set of* synchronizers*; each synchronizer* s *is of the format* $s \in \{?, !\} \times \Sigma$ *where elements in* Σ *represents a synchronizer symbol. Conceptually, "*?*" represents receiving, and "*!*" represents sending.*
- $q_i \in Q_i$ *is the* initial location *of the automaton.*
- $Jump_i = Q_i \times Guards_i \rightarrow Q_i \times Resets_i$ *is the jump from mode to mode where every element in* $Guards_i$ *is of the form* $c_{i_x} \geq k$ *or* $c_{i_x} \leq k$*,* $c \in Q$*, and* $Resets_i$ *is the set of assignments of the form* $c_{i_x} = 0$*.*
- Inv_i *is the set of mode invariants mapping a mode to a subspace of* $R^{|C_i|}$ *indicating the possible clock values to maintain in the mode.*

Fig. 2 shows the semantics for these models. In fig. 2(a) is a Kripke structure $S = \{\{q_0, q_1, q_2, q_3\}, q_0, R, L\}$ where $R = \{(q_0, q_0), (q_0, q_1), (q_1, q_3), (q_3, q_3), (q_3, q_2), (q_2, q_0)\}$, L maps q_0 to set $\{a, b\}$, q_1 to set $\{a\}$, q_2 to set $\{b\}$, and q_3 to the empty set. Fig. 2(b) contains two automata trying to communicate using synchronizers ?`sig` and !`sig`. The inequality in each state is the invariance condition. Synchronizing jumps occur in CTA between different automata with synchronizers with *matching names*. $\mathcal{A}_1$ synchronizes with $\mathcal{A}_2$, jumping from mode q_{11} to q_{12} when it receives signal `sig` and the clock value c_1 is 2. In the context of timed automata, transmission of signals takes no time.

2.3 Specifications in Mathematical Logics

Temporal logic is used to specify the behavior of a system by describing sequences of transitions between states. Specifications can be linear time, where conditions are on a single infinite path of the system, or branching time, where conditions are on the tree structure of all paths of the system. [38] contains a comprehensive review of formal specification languages. The following temporal logics are acceptable by verification engines we used:

PLTL (Propositional Linear Temporal Logic) [30] is a description language for linear time. The PLTL formula $\mathbf{G}(open \rightarrow \mathbf{F}\ closed)$ specifies the requirement that in the state sequence every (operator $\mathbf{G}$) "open" is eventually followed by a (operator $\mathbf{F}$) "closed"[2].

CTL (Computation Tree Logic) [13] and RTCTL (Real-time CTL) [15] are descriptions of branching time temporal logic. TCTL (Timed CTL) [1] is often used in timed models.

[2]Note that our description makes a state with properties both open and closed satisfiable.

2.4 Model Checking

Model checking [12] is a technique for verifying non-terminating (finite) state machines. We apply it on a converted model to see if its behavior conforms to the expectations specified in temporal logic. For details, see [12]. While invariants in general cannot be generated algorithmically using deductive verification, CTL, RTCTL, PLTL, and TCTL model checking is computationally decidable [12, 38], if not solvable in P-time.

3 SR Model Analysis and Conversion

Converting models in the SR domain into Kripke structures requires (1) understanding execution semantics, (2) performing suitable data abstraction, and (3) analyzing layered model structures. We discuss these three tasks below.

3.1 Execution Semantics of FSMActors under the SR Domain

In Ptolemy II, *FSMActors* are extended state machines which record values of variables for each state. They are the basic blocks in verification; in theory, most actors can be represented as finite state machines (see section 4.1 for exceptions). An FSMActor is $A_{fsm} = (Q_{fsm}, P_{fsm}, Para_{fsm}, T_{fsm})$.

- Q_{fsm} is the set of *states*. One state, $q_0 = \{\phi, P_{q_0}, Para_{q_0}, \phi\} \in Q_{fsm}$ with a parameter (`initial`, `true`), is the *initial state* of the FSMActor.
- P_{fsm} is the set of ports.
- $Para_{fsm}$ contains elements of the format $(Var_i, IniVal_i)$, which is the set of *inner variables* and corresponding initial values. We denote $Para_{fsm}.name$ as the *name set* for inner variables.
- $T_{fsm} = \{(p_s, Para_{sd}, p_d) | \, p_s \in P_s, p_d \in P_d, q_s = \{\phi, P_s, Para_s, \phi\} \in Q_{fsm}, q_d = \{\phi, P_d, Para_d, \phi\} \in Q_{fsm}\}$ is the set of *transitions*. $q_s \in Q$ is the *source* state and $q_d \in Q$ is the *destination* state. Each transition has the following properties:
  - exp_{guard} is the *guardExpression*; when its guardExpression is satisfied, a transition triggers. Each of the zero or more subexpressions of exp_{guard}, exp_{guard_i}, puts a constraint on inner variable Var_i, where $1 \leq i \leq |Para_{fsm}|$. In some cases, additional subexpressions exp_{guard_j} ($j > |Para_{fsm}|$) of the form "`PortName_isPresent`" indicate whether a token is visible in the port named `PortName`.
  - act_{output} is the *outputAction*, which specifies destination ports and values. During simulation, each subexpression of act_{output} generates an *event* consisting of a *value* and *time stamp*. FSMActors are zero-delay actors, so events from FSMActors get the current time stamp.
  - act_{set} is the *setAction* which updates the values of inner variables. Each of the zero or more subexpressions act_{set_i} updates the inner variable Var_i, where ($1 \leq i \leq |Para_{fsm}|$).

The semantics of actors in Ptolemy II are defined by the controlling director combined with its *action methods*: preinitialize(), initialize(), prefire(), fire(), postfire() and wrapup(). For details of FSMActors in the SR domain, see [8]; a summary follows.

1. preinitialize() and initialize() are used for initializing; they are invoked only once in the execution.
2. prefire(), fire(), and postfire() are executed once every time the SR director advances time.
   - In prefire(), an actor examines conditions to fire(); FSMActors can always fire().
   - In fire(), an FSMActor tests its exp_{guard} to see if a transition can be triggered. If so, it invokes the transition and executes act_{output}. Subexpressions of exp_{guard} test against the *current* values of inner

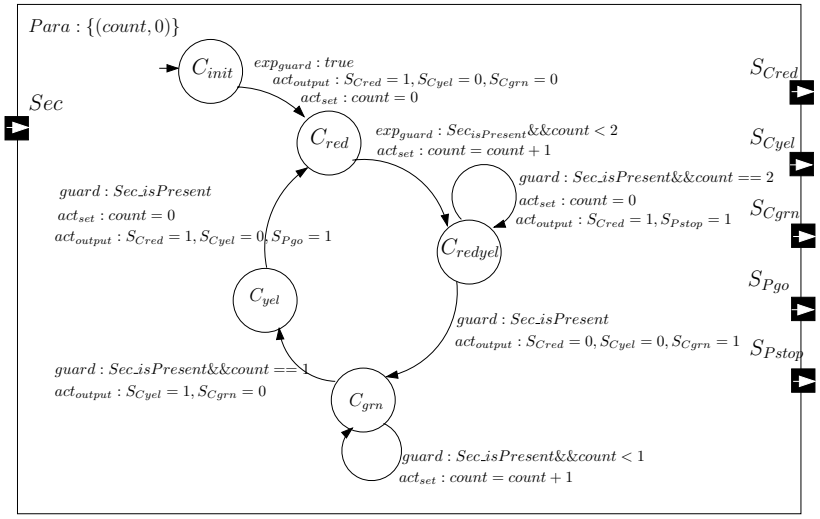


Figure 3: An FSMActor indicating the inner processing of a traffic light.

variables. Subexpressions of the format `X_isPresent` pass when the actor can retrieve a token from the port `X`.

- In postfire(), if the FSMActor triggered a transition in fire(), it executes that transition's act_{set} to update its state. Therefore, in each clock tick, the FSMActor can make at most one transition.

3. wrapup() is invoked once at the end of execution to display results.

3.2 Abstract Interpretation of Variable Domains in FSMActors

When converting an FSMActor $A_{fsm} = (Q_{fsm}, P_{fsm},$
$Para_{fsm}, T_{fsm})$ to a Kripke structure, the number of states in the corresponding Kripke structure may not be $|Q_{fsm}|$; since FSMActors can have inner variables, each in effect is a number of different states, one for each possible combination of inner variable values. To achieve correctness of verification, encoding different values of inner variables into states (and thus APs) is needed. Nevertheless, there is no exact specification on variable domains in a Ptolemy II model; when the variable domain is infinite, abstraction techniques are required to generate a finite partitioning of domains. The following steps describe the function $GVD(A_{fsm}, v, span)$, which generates the abstracted domain for the integer variable v. The FSMActor in fig. 3 is used to illustrate the process.

1. Retrieve the initial value from the parameter set, and scan through the r-values of subexpressions in exp_{guard} and act_{set} for each transition. The minimum and maximum of the variable domain are the minimum and maximum amongst these r-values. In fig. 3, the domain of $count$ is $\{0, 1, 2\}$.

2. Use the user specified constant span to expand the variable domain. If the span is 1, then the expanded domain of $count$ is $\{-3, -2, -1, 0, 1, 2, 3, 4, 5\}$.

3. Use the symbol gt to represent values greater than the domain maximum and the symbol ls to represent values less than the domain minimum. The concept of regions in timed automata [2] inspires the abstract interpretation

of these variables. Thus in the Kripke structure, $\{[count{==}ls],\ [count{==}{-}3],\ [count{==}{-}2],\ [count{==}{-}1],\ [count{==}0],\ [count{==}1],\ [count{==}2],\ [count{==}3],\ [count{==}4],\ [count{==}5],\ [count{==}gt]\}$ is the set of atomic propositions representing the domain of $count$.

By scanning all transitions, we can convert FSMActors to Kripke structures once the variable domain is decided. The *guardExpression* determines the possible variable values that will trigger a transition, and the *setAction* determines the values of inner variables after the transition triggers. The algorithm follows:

Convert_FSMActor_To_Kripke_Structure_SR (A_{fsm}, $span$){
/* $A_{fsm} = (Q_{fsm}, Para_{fsm}, P_{fsm}, T_{fsm})$ */
Let the set of atomic propositions AP be $(\bigcup_{q \in Q_{fsm}} [state_{fsm}{==}q])$
$\bigcup (\bigcup_{v \in Para_{fsm}.name,\ val \in GVD(A_{fsm},v,span)} [v == val])$;
Let the set of states in the Kripke structure be
$S = (\bigwedge_{q \in Q_{fsm}} \mathbf{2}^{\{[state_{fsm}==q]\})}$
$\bigwedge (\bigwedge_{v \in Para_{fsm}.name,\ val \in GVD(A_{fsm},v,span)} \mathbf{2}^{\{[v==val]\})}$;
/* A state in the Kripke structure is a tuple:
$([state_{fsm} == state_1], \ldots, [state_{fsm} == state_n],$
$[v_1 == val_{1,1}], \ldots, [v_1 == val_{1,|GVD(A_{fsm},v_1,span)|}],$
$\ldots$
$[v_{|Para_{fsm}|} == val_{|Para_{fsm}|,1}], \ldots,$
$[v_{|Para_{fsm}|} == val_{|Para_{fsm}|,|GVD(A_{fsm},v_{|Para_{fsm}|},span)|}])$
where each term can be either **0** or **1**.
$val_{i,j}$ represents the j-th possible value of variable i.*/
Let the mapping function be $L := S \rightarrow S$;
Let the initial state s_0 of the Kripke structure be
$([state_{fsm} == state_1], \ldots, [state_{fsm} == state_{|Q_{fsm}|}],$
$[v_1 == val_{1,1}], \ldots, [v_1 == val_{1,|GVD(A_{fsm},v_1,span)|}],$
$\ldots$
$[v_{|Para_{fsm}|} == val_{|Para_{fsm}|,1}], \ldots,$
$[v_{|Para_{fsm}|} == val_{|Para_{fsm}|,|GVD(A_{fsm},v_{|Para_{fsm}|},span)|}])$
where
(1) $[v_i == val_{i,j}] =$ **1** if $val_{i,j} == IniVal_i$ and **0** otherwise
(2) $[state_{fsm} == state_i] =$ **1** if $state_i == q_0$ and **0** otherwise;
Let $R = \phi$ be the set of transitions;
$\forall r = (s, s') \in S \times S$, $R = R \cup \{r\}$ if the following holds.
(1) $\exists$**1** $i, 1 \le i \le |Q_{fsm}|$ s.t. in s,
entry $[state_{fsm} == state_i] ==$ **1**,
(2) $\exists$**1** $i', 1 \le i' \le |Q_{fsm}|$ s.t. in s',
entry $[state_{fsm} == state_{i'}] ==$ **1**,
(3) $\forall j, 1 \le j \le |Para_{fsm}|$,
$\exists$**1** $k, k',\ 1 \le k, k' \le |GVD(A_{fsm}, v_j, span)|$
s.t. in s, entry $[v_j == val_{j,k}] ==$ **1** and in s',
entry $[v_j == val_{j,k'}] ==$ **1**,
(4) Based on information in (1),(2),(3),
$\exists t = (p_s, Para_{sd}, p_d) \in T_{fsm}$:

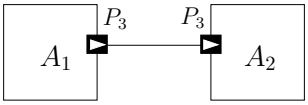


Figure 4: Two connected ports with matching names.

(a) $p_s \in P_s, p_d \in P_d$, $state_i = \{\phi, Para_s, P_s, \phi\} \in Q_{fsm}$,
$state_{i'} = \{\phi, Para_d, P_d, \phi\} \in Q_{fsm}$,
(b) $\forall j, 1 \leq j \leq |Para_{fsm}|, exp_{guard_j}(val_{j,k}) == \mathbf{1}$,
and $act_{set_j}(val_{j,k}) == val_{j,k'}$;
return $(S, \{s_0\}, R, L)$;
}

This method leads to an over-approximation of the system behavior with no false positives[3]. If the variable is closed within the value domain generated by steps 1 and 2, the introduction of gt and ls has no effect on the correctness of verification.

3.2.1 Interactions between FSMActors

The interactions between actors must be considered when converting a model where FSMActors communicate with each other into a Kripke structure. An FSMActor is a zero-delay actor; based on the synchrony hypothesis [6], actors are immediately notified when sent an event and further actions may occur. Therefore, transitions in different modules may depend on each other. Our tool automatically establishes transition dependencies among FSMActors when converting to Kripke structures.

Formally, for two actors $A_{fsm_1} = (Q_{fsm_1}, P_{fsm_1}, Para_{fsm_1}, R_{fsm_1})$and $A_{fsm_2} = (Q_{fsm_2}, P_{fsm_2}, Para_{fsm_2}, R_{fsm_2})$, the set of state spaces S_{12} is the product of the state spaces for each Kripke structure, i.e.,
$S_{12} = \bigwedge_{q_1 \in Q_{fsm_1}} 2^{\{[state_{fsm_1} == q_1]\}} \bigwedge_{q_2 \in Q_{fsm_2}} 2^{\{[state_{fsm_2} == q_2]\}}$
$\bigwedge_{v \in Para_{fsm_1}.name,\ val_1 \in GVD(A_{fsm_1}, v_1, span)} 2^{\{[v_1 == val_1]\}}$
$\bigwedge_{v_2 \in Para_{fsm_2}.name,\ val_2 \in GVD(A_{fsm_2}, v_2, span)} 2^{\{[v_2 == val_2]\}}$.

Similarly, the set of APs is the union of the set of APs for each Kripke structure. The set of transitions R_{12} may include cases with ports with *matching names*; we assume the connected ports have the same name, and unconnected ports have different names[4] (see fig. 4). The conversion algorithm is similar to the one in sections 3.2, so we only describe the matching name section below.

Let R be the set of transitions;
$r = (s, s') \in S_{12} \times S_{12}$, $R = R \cup \{r\}$ if following conditions hold.
(1) $\forall p,\ p = 1, 2, \exists \mathbf{1}\ i_p, 1 \leq i_p \leq |Q_{fsm_p}|$
s.t. in s, entry $[state_{fsm_p} == state_{i_p}] == \mathbf{1}$,
(2) $\forall p,\ p = 1, 2, \exists \mathbf{1}\ i'_p, 1 \leq i'_p \leq |Q_{fsm_p}|$

[3]This is true when sub-expressions of exp_{guard} are simple, meaning that they are of the format "$var\ (\leq, \geq, <, >, ==)\ const$". For complex expressions involving two or more variables, it can be true only when gt and ls are never reached throughout the computation.

[4]This can be further improved by mechanisms of detecting connected ports, but difficulties would later emerge when converting in the DE system.

s.t. in s', entry $[state_{fsm_p} == state_{i'_p}] == \mathbf{1}$,
(3) $\forall p,\ p = 1, 2, \forall j_p, 1 \leq j_p \leq |Para_{fsm_p}|$,
$\exists \mathbf{1}\ k_p, k'_p,\ 1 \leq k_p, k'_p \leq |GVD(A_{fsm_p}, v_{j_p}, span)|$
s.t. in s, entry $[v_{j_p} == val_{j_p,k_p}]$==**1**;
in s', entry $[v_{j_p} == val_{j_p,k'_p}]$==**1**,
(4) Based on information in (1),(2),(3),
$\exists t_1 = (p_{s_1}, Para_{sd_1}, p_{d_1}) \in T_{fsm_1}$,
$\exists t_2 = (p_{s_2}, Para_{sd_2}, p_{d_2}) \in T_{fsm_2}$:
(a) $p_{s_1} \in P_{s_1}, p_{d_1} \in P_{d_1}$, $state_{i_1} = \{\phi, Para_{s_1}, P_{s_1}, \phi\}$ $\in Q_{fsm_1}$, $state_{i'_1} = \{\phi, Para_{d_1}, P_{d_1}, \phi\} \in Q_{fsm_1}$,
(b) $p_{s_2} \in P_{s_2}, p_{d_2} \in P_{d_2}$, $state_{i_2} = \{\phi, Para_{s_2}, P_{s_2}, \phi\}$ $\in Q_{fsm_2}$, $state_{i'_2} = \{\phi, Para_{d_2}, P_{d_2}, \phi\} \in Q_{fsm_2}$,
(c) $\forall j_1, 1 \leq j_1 \leq |Para_{fsm_1}|, exp_{guard_{j_1}}(val_{j_1,k_1}) == \mathbf{1}$, and $act_{set_{j_1}}(val_{j_1,k_1}) == val_{j_1,k'_1}$,
(d) $\forall j_2, 1 \leq j_2 \leq |Para_{fsm_2}|, exp_{guard_{j_2}}(val_{j_2,k_2}) == \mathbf{1}$, and $act_{set_{j_2}}(val_{j_2,k_2}) == val_{j_2,k'_2}$,
(e) $\exists\ exp_{guard_{k_1}}, k_1 > |Para_{fsm_1}|$ of the textual format `PortName_isPresent`;
iff $\exists\ act_{output_{k_2}}$, a subexpression of act_{output_2}, of the textual format `PortName=IntegerValue`;

Our current implementation can process more than two FSMActors.

3.3 Hierarchical Compositionality

In Ptolemy II, the *HigherOrderActor* library provides support for tree structured, compositional models. The *ModalModel*, which can be viewed as an FSMActor with refinement (or a subsystem) in each state, is especially significant. If each subsystem uses continuous dynamics, the ModalModel describes discrete jumps between continuous dynamics, which is commonly used for hybrid modeling. Formally, a ModalModel $A_{ModalModel}$ consists of (1) $A_{FSMDirector}$ and (2) $A_{fsm_{controller}}$, where $A_{FSMDirector}$ is a director with finite state machine semantics, and an FSMActor $A_{fsm_{controller}} = (Q_{fsm}, P_{fsm}, Para_{fsm}, T_{fsm})$ is the *controller*. The refinement of a state is defined in the state $q \in Q_{fsm}$ itself.

When the controller of a ModalModel in the SR domain invokes fire() on an SR refinement, the following occurs:

1. The FSM director $A_{FSMDirector}$ transfers the input tokens from the outside domain to $A_{fsm_{controller}}$ and the refinement of its current state.
2. The refinement of the current state fires.
3. If a transition triggers, its outputActions execute.
4. Any output tokens produced by the mode controller or the refinement are transferred to the outside domain.

Fig. 5 illustrates this process. In fig. 5(a), state C_1 in the ModalModel has a refinement with SR semantics. When executing (shown in fig. 5(b)), the controller invokes the firing of the refinement, which enables the transition of subsystems. Fig. 5(c) shows the corresponding transition in the converted Kripke structure. In a ModalModel, the controller's state and the states of FSMActors in the refinement update in the same tick when they postfire(), so

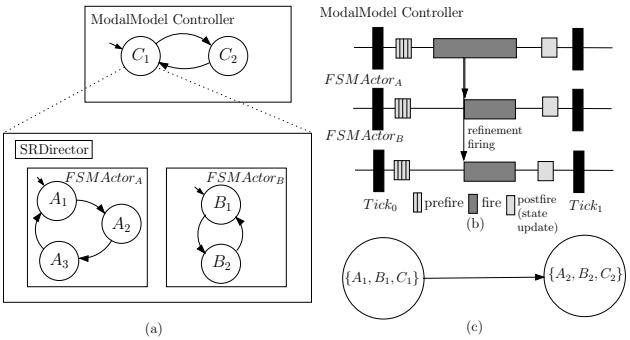


Figure 5: (a) A ModalModel with SR refinement (b) The starting execution trace based on the semantics of ModalModel and SR (c) The corresponding transition in the converted Kripke structure.

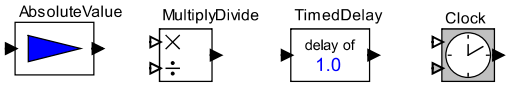


Figure 6: Two functional actors (AbsoluteValue, MultiplyDivide) and two actors (TimedDelay, Clock) with real-time properties.

the controller and refinement constitute a product state space; the semantics of the ModalModel guarantees that the controller and actors in refinements can make one move together between consecutive clock ticks[5].

4 From the SR Domain to the DE Domain

4.1 Actors beyond State Machines

FSMActors (or ModalModels) in the SR domain are suitable for verification because only sequence ordering is relevant; time is abstracted. However, Ptolemy II allows designers to construct actors not representable as FSMActors, which are therefore not suitable for verification in SR. The following summarizes them:

- **(Actors with purely mathematical operations)** Purely functional actors, those that receive input i and generate output $f(i)$, are stateless, and therefore would not appear in the product state space. Examples include actors in the *Math* actor library (see the first two actors in fig. 6). These actors greatly influence transitions and inner state values, though, so we must be careful when using them in designs we wish to verify.
- **(Actors with real-time properties)** Actors in the discrete event (DE) domain often have explicitly stated timing properties. For example, the *TimedDelay* actor (the third actor in fig. 6) postpones an incoming event a certain amount of time, and the *Clock* actor (the fourth actor in fig. 6) sends events with a certain period under a fixed or infinite number of total firings.

[5]It is possible to design another modal controller with different semantics; for example, the update of state space of the subsystem occurs one tick later than the update of controller. In this way, the execution semantics must be reinvestigated for correct model conversion.

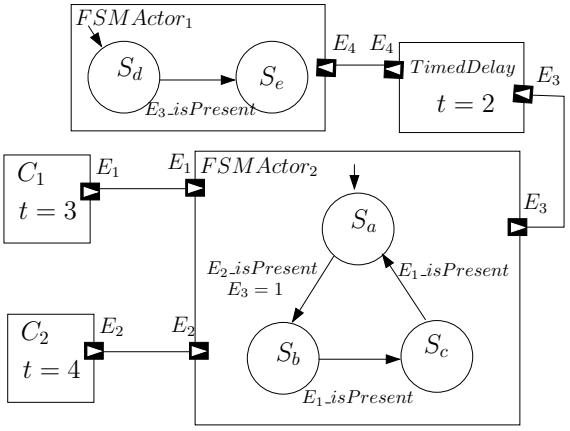


Figure 7: Ptolemy II description of two Clock actors, two FSMActors, and one TimedDelay actor (t=2).

- In Ptolemy II, the DE domain can be viewed as a generalization of the SR domain [28], with a global clock visible to the whole system. Unlike the SR domain, clock readings in the DE domain are arbitrary non-negative real numbers; *super-dense time tags*[6] in the DE domain are equivalent to ticks in the SR domain.
- Note that it is not always possible to use discrete clocks (SR semantics) for these actors. Consider a system having the *Zeno* property, meaning that time converges to a certain time t. If a TimedDelay actor has an event to be fired after t, then it is impossible to use a discrete-clock to represent the advance of time for the actor.

4.2 Converting DE Systems with CTA

For systems with actors with real time properties, we use Communicating Timed Automata, mathematical models with greater expressive power.

4.2.1 Converting FSMActors into CTA

We must make four considerations when keeping an FSMActor's semantics in the DE domain the same as those in the SR domain.

- (Passiveness of FSMActors) In the DE domain, there is no 'tick' explicitly specified; an FSMActor is '*passive*', meaning that it tries to perform the transition when the super-dense time advances (which is invoked by other actors). Therefore, whenever '*active*' actors that send events with real time stamps (and let time advance), FSMActors in the system should always be notified so that they can invoke their transitions.
- (From super-dense time to dense time) Because time is super-dense in the DE domain, semantics of an FSMActor indicate that at every physical time τ, it may perform more than one transition. The challenge now is to describe super-dense time models (DE) with dense time models (CTA). One intuitive approach is to introduce additional synchronizers for ticks; an active actor notifies an FSMActor by using the synchronizer $tick_+$ for

[6]Super-dense time is of the format $T = R_+ \times N$, where R_+ is the set of non-negative numbers, and N is the set of natural numbers. For a tag $t = (\tau, n) \in T$, τ is the physical time, and n is the ordering of events that occurs in the physical time [28].

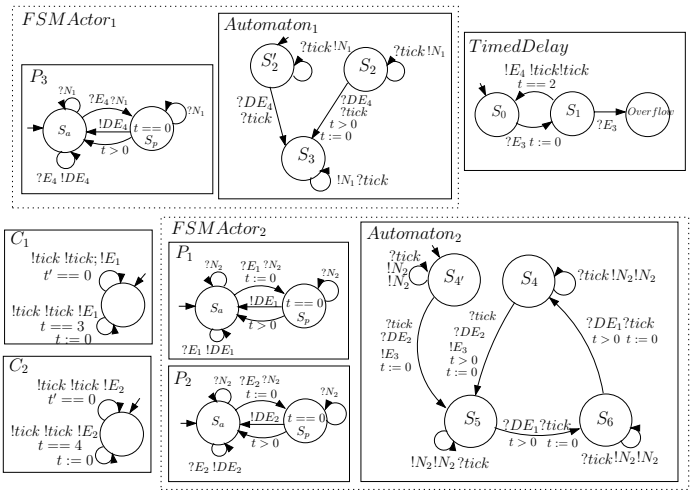


Figure 8: CTA description of two Clock actors, two FSMActors, and one TimedDelay actor (buffer size=1, t=2).

events happening in time $(\tau, 1)$, synchronizer $tick_{++}$ for events happening in time $(\tau, 2)$, and so on. Unfortunately, this approach can not guarantee that for all execution traces in the converted CTA, at time τ every $tick_{+}$ comes before every $tick_{++}$, when there is no causality between two events. To solve this, our current implementation simply disallows a system to have the chance to advance the time from (τ, n) to $(\tau, n+1)$ - this only happens when an event is passed through the TimedDelay actor with zero delay.

- (One step at one time instance) The semantics of an FSMActor in SR indicate the fact that at every tick, it should only perform one transition. Therefore, we need to introduce an additional timer; when a transition is made, set the timer to zero, and make sure that when next transition occurs, the value of the timer is greater than zero.
- (Separated CTA construction for ports) If an FSMActor receives events from other actors, we need to construct a CTA for each of the ports in the FSMActor with two states indicating *present* and *absent*; tokens may not accumulate in a port of an FSMActor in the DE domain[7].

We illustrate the above concepts by examples. The content in fig. 7 is an actor $FSMActor_2$ receiving events from two clock actors C_1 and C_2, and another actor $FSMActor_1$ receives events from $FSMActor_2$ via a TimedDelay actor with t=2. Fig. 8 is its corresponding CTA.

- For all states in the $Automaton$ of the FSMActor, every transition should be bounded by a synchronizer $?tick$; other active actors would send $!tick$ to trigger the transition. An actor sends $!tick$ when the time tag is of the format $(\tau, 0)$.
- When a Clock actor sends an event, the port automaton updates its memory indicating whether the token is available at the port. By introducing an additional synchronizer N_2 in $FSMActor_2$, P_1, and P_2, we can understand the impact of the synchronizer $tick$.
  - If a tick is triggered by another unconnected active actor (for example, an irrelevant C_3 actor), but in the original FSMActor no transition other than the stationary move can be enabled, then the self loop in $Automaton_2$ would be executed and sends synchronizers $!N_2!N_2$ to notify port actors to perform a

[7]For other DE semantics where tokens can accumulate in the port, the CTA representation of a port automaton should (1) be limited with a bounded buffer size and (2) provide an additional state for buffer overflow.

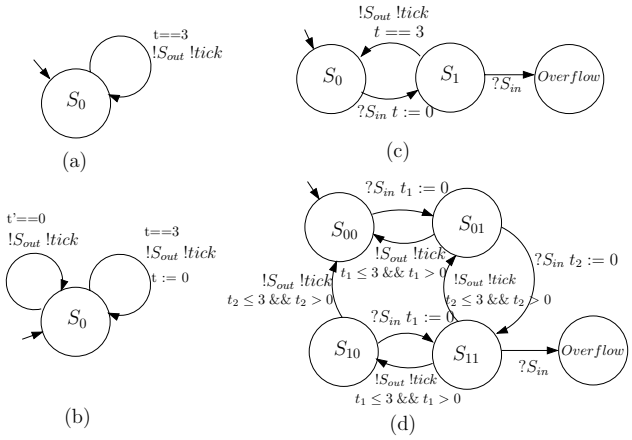


Figure 9: CTA representation of real-time actors in Ptolemy II.

stationary move (the self loop with synchronizer $?N_2$).

- If a tick is triggered by a connected active actor, but no transition other than the stationary move can be enabled, then the synchronizer $!N_2$ would notify the port machine to set up the token to be present.
- If a tick is triggered by a connected active actor, and a transition can be triggered, at this tick, the transition has been decided. We thus let the port automaton stay in the absent state S_a, since when next tick starts, the status is absent.

For example, when time is zero, C_1 and C_2 both send events with time stamp (0,0). P_1 indicates the token to be present and moves from S_a to S_p, while P_2 stays in S_a and $Automaton_2$ takes the transition and moves to S_6. Note that it is guaranteed not to take two transitions in (0,0) because in $Automaton_2$, transitions with the synchronizer $?tick$ are guarded with $t > 0$ (except the transition from the initial state; we thus need to introduce an additional initial state). The next tick is (2,0), which is induced by the TimedDelay actor with t=0. At (2,0), $Automaton_1$ moves to state S_3.

4.2.2 Converting Real-time Actors into CTA

The conversion for real-time actor classes may vary due to their properties. We list out some actor classes currently implemented, and use fig. 9 to describe the converted communicating timed automata. Without loss of generosity, we assume the input is S_{in}, the output is S_{out}, and the notification signal for passive actors is $tick$.

- A *SingleEvent* actor sends an event at a time specified by the parameter t. Fig. 9(a) is a SingleEvent actor with $t = 3$.
- A *Clock* actor periodically sends an event to its output port with period t for k times. Both t and k are specified by parameters. The value k can be `INFINITY`, meaning that it would continuously sends events without halting. Fig. 9(b) is a Clock actor with $t = 3$ and $k =$ `INFINITY`. Note that additional clock t' must be used to capture the fact that it sends the first event at time zero.
- A *TimedDelay* actor postpones an event with a time specified by the parameter t. Another parameter s is used to specify the buffer size of the input port. Fig. 9(c) is a TimedDelay actor with buffer size $s = 1$. When the actor receives an event, it goes to state S_1. Note that the size of the buffer must be constrained by users, and more clock variables will be introduced when the size of the buffer grows.
- A *NonDeterministicTimedDelay* actor postpones an event with time at most t and guarantees the non-zero delay.

Another parameter s is used to specify the buffer size of the input port. Fig. 9(d) is a NonDeterministicTimedDelay actor with buffer size 2. Note that translating the actor to CTA may lead to exponential growth for the size of control locations, since the ordering of emitting signals can be arbitrary.

Based on statements described above, we are able to establish the conversion process which separately performs the conversion for each component in the DE domain (preprocessing is needed to detect dependencies of ports) and form the CTA with semantics preserved.

5 Designing for Verification

When the interactions between a verification model and its environment serve as a guide for how an implementation model will interact with its sensors and actuators, the verification model can be reused as the basis of the implementation model, resulting in less work for the designer and less potential for errors. In this section, we explain this methodology and give an example of how to follow it to design a model for both verification and implementation.

5.1 Methodology

Step 1: Evaluate Requirements A designer is given a number of requirements for an embedded system, some of which are safety critical which require verification. The first step is to identify safety critical requirements.

Step 2: Establish a Contract and Design the Environment Physical, continuous environments contain far too much complexity for efficient verification, so in this step we establish an abstraction with the safety critical requirements of the system in mind. This abstraction, which should be rigorously defined, will establish a blueprint for the model of the environment.

Step 3: Design and Verify a Controller In this step, the actual controller is modeled, connected to the environment, and verified as fulfilling all safety critical requirements.

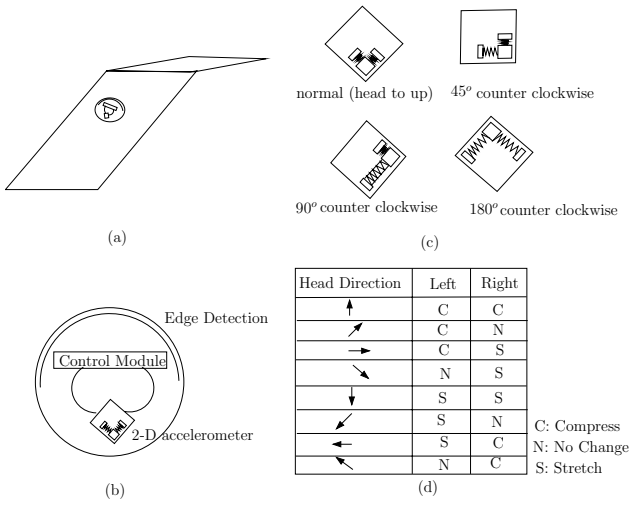
Step 4: Follow the Contract to Design Sensors and Actuators The controller has only been verified to handle certain value ranges, so when it is used as an implementation controller it can only be expected to handle those abstracted values. In this step, the contract defined in Step 2 will help define wrappers for the sensors and actuators. These wrappers will translate the raw data used by sensors and actuators to the discrete data values which the controller has been verified to handle. Once complete, the controller is disconnected from the environment and connected to the sensors and actuators.

Step 5: Add Non-critical Requirements The non-critical requirements are now added to the controller, so all requirements are fulfilled.

Step 6: Automatic Code Generation To complete the process, code is generated for the controller and the sensors and actuators and then put on the target platform.

5.2 Example: Hill-Climbing Robotic Design

A robot is an embedded system consisting of sensors, actuators and controls, that tries to *achieve* a certain goal. [29] is a good introduction to robotics. In this example, we illustrate the above design strategy by constructing a robot capable of climbing a fixed width slope (fig. 10(a)). iRobot Create [24] is our target platform. The robot has two sensors: edge detectors in the front half, and a two dimensional accelerometer (fig. 10(c)). We strive to design the system and generate real control software such that (1) the robot never falls from the edge of the slope, (2) the robot reaches the flat surface above the slope, (3) the robot's LEDs blink, and (4) its speaker makes sound.



Head Direction	Left	Right
↑	C	C
↗	C	N
→	C	S
↘	N	S
↓	S	S
↙	S	N
←	S	C
↖	N	C

Figure 10: A robot climbing a hill (a), the accelerometer attached to the robot (b), the hypothetical inner structure in the 2-D accelerometer (c), and the abstracted signal emissions for all robot heading directions (d).

Step 1: Evaluate Requirements That the robot never falls off the edge and that it reaches the top of the slope are safety critical, and should be verified. Blinking LEDs and noisy speakers can be added later.

Step 2: Establish a Contract and Define the Environment In order to perform verification, we must discretize the environment at a manageable granularity. The environment sends data to the robot's sensors based on the robot's location and orientation, both of which need to be discretized. Examining the iRobot and its edge detectors, we discover that moving in 1 cm increments will ensure that the robot will never jump from safety to danger in one time step. To discretize orientation, we think of the 2-D accelerometer as two 1-D accelerometers placed orthogonally, each of which can be viewed as a point mass attached to a spring. When horizontal, the spring has no compression or extension; when upward facing, the gravitational force compresses it; when downward facing, the gravitational force stretches it. Therefore, the robot can only face eight directions, defined by the accelerometer readings shown in (fig. 10(d)). For example, when upward facing, both springs are compressed (fig. 10(c)).

For verification, we use an FSMActor to represent the environment. The location and the angle of the robot are stored as inner variables of the FSMActor, with initial values defined in the parameter set. When a signal is sent by the controller, like driveForward, the inner variables and state of the environment are updated, and new information is sent to the controller. The environment sends the controller a warning with an L or R (for left or right, respectively) when it is close to an edge and a pair from the set $\{C, N, S\}$ for orientation.

The following is our contract:

- The robot is only able to rotate in place or drive in a straight line.
- The robot drives either 1 cm if horizontal or vertical, or 1 cm horizontally and 1 cm vertically if diagonal.
- The robot always faces one of eight directions, defined by pairs of values from the set $\{C, N, S\}$.

Step 3: Design and Verify a Control We next design the controller. When complete, we link the controller to the environment, apply automatic conversion, and perform verification on the system.

Step 4: Follow the Contract to Design Sensors and Actuators Once the controller is verified to satisfy the specifications, we generate wrappers for sensors and actuators, making sure they satisfy the requirements of the contract. We then disconnect the controller from the environment used for verification and attach it to the wrappers. The

wrappers will convert raw data to values the controller understands as follows:

- The reading of an actual 1-D accelerometer is a real number which we lump into ranges corresponding to the set $\{C, N, S\}$.
- A *Drive* signal sent to the wheels causes the robot to drive 1 cm if it is horizontal or vertical or $\sqrt{2}$ cm if diagonal.

Step 5: Add Non-critical Requirements To ensure that the robot starts with the orientation specified as an initial condition, a pre-operation of the *Init* state performs in place rotation. We also add functionality for blinking LEDs and noisy speakers.

Step 6: Automatic Code Generation With an implementation controller that has passed critical verification and fulfils all requirements, we generate code for the target platform.

6 Implementation

6.1 MathematicalModelConverter Actor

We test our concepts in the Ptolemy II architecture, and develop algorithms to perform the conversion process. For the SR domain, we translate models into formats acceptable by Cadence SMV (also NuSMV)[8]; for the DE domain, we translate models into formats acceptable by RED [37]. An experimental *MathematicalModelConverter* actor class allows users to input temporal formula and perform one-click conversion and verification.

Limitations In our current implementation, there are some existing restrictions in the MathematicalModelConverter actor class.

- When the conversion process is applied in the SR domain, an actor must be either an FSMActor or an ModalModel. Otherwise it would be automatically translated as an external signal which is present at every tick.
- The freedom of design is constrained by the name-matching of ports.
- The introduction of gt and ls is only experimented in the SR domain; in the DE conversion process users must understand the domain for inner variables.
- We do not experiment with the hierarchical conversion in the DE domain.

However, our current work is still complete for a subset of system construction. We would add up functionalities to surpass the above limitations soon.

6.2 Integrating Model Checkers using the EmbeddedCActor

By incorporating external verification software into Ptolemy II, the entire process of model design, verification, and code generation can be done through Ptolemy II. Using the EmbeddedCActor, an actor that allows the execution of arbitrary C or C++ code in a model, we can incorporate any verification software written in C or C++.

We incorporate the NuSMV software into an EmbeddedCActor, thereby incorporating the entire verification process into Ptolemy II. The steps for incorporating verification software are as follows:

Step 1: Create a static library For NuSMV, the desired function is the main function. To make the main function easily accessible, we wrap its contents as a new function in a separate file, then add that file to the static library that NuSMV automatically creates and links to the main file. We also face the difficulty of conflicting files called `config.h`, which we overcome by renaming NuSMV's `config.h` to `NuSMVconfig.h`.

Step 2: Link the library to an EmbeddedCActor By linking the library to an EmbeddedCActor and including the headers needed by the library, the code in the EmbeddedCActor can access the contents of the library.

[8] Since these tools support modular construction and verification, our latest implementation utilizes this benefit.

Step 3: Access the generated notation file When verifying a model, the model is translated into a formal mathematical notation which is stored in a separate file; the file name can either be passed to the EmbeddedCActor through a port, or can be set in a parameter. We choose to set a parameter with the path to the file.

Step 4: Prepare arguments In some cases, this step may be trivial, but because we use the NuSMV main function, we need to simulate calling the function from the command line, which requires some preparation. We need to decide which command line options to use, then create artificial `argc` and `argv` variables to pass to the verification function.

Step 5: Call the desired functions For NuSMV, we simply have to call the function containing the content of the original main function. NuSMV prints its output, so after calling the function we called `fflush(0)` to ensure that the output from verification shows up on the console.

The total procedure takes very little code.

- Wrapping the main function involves copying the existing code into a new file; only a few lines of code are needed.
- Changing the name of `config.h` requires changing one line of code in many files, which was easily accomplished using search and replace.
- The EmbeddedCActor itself comprise approximately 50 lines, including those needed to include headers and link to libraries.

7 Case Studies

7.1 Traffic Light System with Hierarchical Structures

In the case study, the traffic light model specified in [7] is investigated. We construct a simplified traffic light system in the SR domain, where the system consists of one car light and one pedestrian light. Our design should make it impossible to have the car light and pedestrian light be green at the same time (this might lead to accidents). We perform the automatic conversion of the SR system, and are able to test whether our design satisfies the specification:

!EF (`TrafficLight.CarLightNormal.state` = `Cgrn` &
`TrafficLight.PedestrianLightNormal.state` = `Pgreen`)

The result provided by the model checker SMV indicates the satisfiability of the specification.

7.2 Extended Experiment in the DE Domain

Buffer overflow detection has been an important issue in the design of embedded systems. Our conversion process enables us to configure parameters of buffer size for each component, and use the model checker RED to detect the undesired behavior. More precisely, the *buffer overflow detection* property can be stated as the *reachability* property of the system by the following formula (specified in the language acceptable by RED) `exists i:i>=1, (Overflow[i])`. We can also describe more interesting properties using TCTL.

However, we find that our converted model poses great challenges for real time model checkers. RED is a memory efficient and relatively fast tool, but the computation time for small examples converted by our algorithm remains large. Since for BDD-like structures, variable ordering is important, it may require a tight integration between the conversion process and lower layer verification engine so that the converted model can have smaller memory space during the verification process.

7.3 Designing the Robotic Controller Satisfying Hill Climbing and Edge Avoidance Properties

For the example in section 5.2, we perform model checking on the model and the environment. One desired specification is the eventuality of the robot to reach the top surface. The specification can be described as follows. **AF**(`Slope.state` = `Goal`). The result indicates that our tool is applicable for large scale conversion. We later follow the process and generate the control software for the robot[9].

8 Related Work

In academia, algorithmic model checking began by checking the correctness of hardware and then moved to software. Advances in real code verification include JavaPathFinder [18], BLAST [21], CBMC [10] and SLAM [3].

Researchers primarily focus on verification algorithms in the domain of embedded and real time systems, leading to models based on theoretical units rather than engineering units. For example, hybrid automata [19] are commonly studied for modeling and verifying embedded systems. Uppaal [25], among other tools based on mathematical models, is capable of modeling and verifying real time systems using timed automata.

Giotto [20], Metropolis [4], and BIP [5] are languages or design environments capable of modeling embedded systems developed in academia.

Various tools have emerged from industry. STATEMATE [34] is based on hierarchical state charts and is capable of verifying state machine systems. The SCADE Suite [31], a mature tool developed by Esterel-Technologies, uses synchronous reactive language for the design of safety critical systems. The Simulink Design Verifier [32] can perform formal analysis on Simulink and Stateflow models.

[26] and [23], two positional papers on design challenges in Cyber Physical Systems and embedded systems, respectively, are important as guidance for future work.

9 Conclusion and Future Work

We summarize our main contributions as follows.

- Our concept of applied verification narrows the gap between designing systems and verifying them by shielding designers from the details of verification. Simultaneously, designers become more productive and errors become less common.
- We have developed a mapping between commonly used Ptolemy II actors and mathematical models used in verification, allowing for the automatic conversion of some Ptolemy II models in the SR and DE domains.
- We describe our methodology of designing for verification, which allows for models under verification to be reused for final implementation.
- We describe our strategy for integrating verification engines into Ptolemy II.

By combining the existing simulation framework in Ptolemy II with a system for automatically converting simulation models to mathematical models, we offer the beginning of a solution for checking the correctness of heterogenous system design: correctness of data can be established by running simulations, while control flow can by checked using formal verification.

Our work reveals interesting problems for future study:

[9] We adapt the existing code generation framework in the SDF [27] domain to generate C++ code; SR and SDF are the same in this example. Nevertheless, we need to slightly modify the control without changing the functionality to perform SDF code generation. For example, we change EdgeLeft_isPresent and NoEdgeLeft_isPresent in every guardExpression in the control into EdgeLeft==1 and EdgeLeft==0, and later remove the port NoEdgeLeft.

- By providing the foundations for relating Ptolemy II models to mathematical models used for verification, we open the doors for applying more powerful verification techniques. For example, we could use rate control analysis for parametric safety analysis.
- Our analysis technique is powerful enough to use for other models of computation, e.g. the *distributed discrete event* (DDE) domain. We also hope to use hybrid automata as our underlying mathematical models, giving us more expressive power.
- We expect to use more compact description language to describe the behavior of the environment (e.g., Ptylon), and investigate possibilities applying Giotto to describe contracts of the environment.

Acknowledgments

We thank Latronico Elizabeth, Thomas Feng, Issac Liu, Prof. Farn Wang, Prof. Thomas A. Henzinger, and Prof. Daniel Kröning for very helpful suggestions.

References

[1] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking for real-time systems. In *Logic in Computer Science (LICS)*, pages 414–425, IEEE, 1990.

[2] R. Alur, and D.L. Dill. A Theory of Timed Automata. In *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] T. Ball, and S.K. Rajamani. The SLAM toolkit. In *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 200–264, Springer-Verlag, 2001.

[4] F. Balarin, et al. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, IEEE, 2003.

[5] A. Basu, et al. Modeling Heterogeneous Real-time Components in BIP. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–12, IEEE, 2006.

[6] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[7] C. Brooks, T.H. Feng, E.A. Lee, and R. von Hanxleden. Multimodeling: A Preliminary Case Study. Technical Report UCB/EECS-2008-7, University of California, EECS Department, 2008.

[8] C. Brooks, et al. Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2007-7, University of California, EECS Department, 2008.

[9] A. Cimatti et al. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 259–364, Springer-Verlag, 2002.

[10] E.M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-CPrograms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176, Springer-Verlag, 2004.

[11] E.M. Clarke, and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In: *Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71, Springer-Verlag, 1981.

[12] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[13] E.A. Emerson, and J.Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of ACM*, 33(1):151–178, ACM, 1986.

[14] J. Eker, et al. Taming Heterogeneity—the Ptolemy Approach. *Proceedings of the IEEE*, 91(2):127–144, IEEE, 2003.

[15] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *International Journal in Real Time System*, 4:331–352, 1992.

[16] H. Giese, et al. Model-based Engineering of Embedded Real-time Systems. In *Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI)* , volume 7451 of *Dagstuhl Seminar Proceedings*, 2007.

[17] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, IEEE, 1999.

[18] K. Havelund, and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, Springer, 2000.

[19] T.A. Henzinger. The Theory of Hybrid Automata. In *Logic in Computer Science (LICS)*, pages 278–292, IEEE, 1996.

[20] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In *Embedded Software: First International Workshop (EMSOFT)*, volume 2211 of *LNCS*, pages 166–184, Springer-Verlag, 2001.

[21] T.A. Henzinger, et al. Software Verification with BLAST. In *International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *LNCS*, pages 235–239, Springer-Verlag, 2003.

[22] T.A. Henzinger. Games, time, and probability: Graph models for system design and analysis. In *International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 4362 of *LNCS*, pages 103–110, Springer-Verlag, 2007.

[23] T.A. Henzinger, and J. Sifakis. The discipline of embedded systems design. *Computer*, 40(10):36–44, IEEE, 2007.

[24] iRobot Create: `http://www.irobot.com/create/`

[25] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, Springer, 1997.

[26] E.A. Lee. Cyber Physical Systems: Design Challenges. Technical Report UCB/EECS-2008-8, University of California, EECS Department, 2008.

[27] E.A. Lee and T.M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, IEEE, 1995.

[28] E.A. Lee, and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 114–123, ACM, 2007.

[29] M.J. Mataric. The Robotics Primer (Intelligent Robotics and Autonomous Agents). MIT Press, 2007.

[30] A. Pnueli. The temporal logic of programs. In *Symposym Foundations of Computer Science (FOCS)*, pages 45–57, IEEE, 1977.

[31] SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite/`

[32] Simulink Design Verifier. `http://www.mathworks.com/products/sldesignverifier/`

[33] I. Sommerville. Software Engineering. Addison Wesley, 2006.

[34] STATEMATE. `http://modeling.telelogic.com/products/statemate/`

[35] J. Sztipanovits, and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–112, IEEE, 1997.

[36] J. Tsay, and E.A. Lee. A code generation framework for Java component-based designs. In *International conference on Compilers, architecture, and synthesis for embedded systems (CASES)*, pages 18–25, ACM, 2000.

[37] F. Wang. Efficient Data Structure for Fully Symbolic Verification of Real-Time Software Systems. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*, pages 142–156, Springer-Verlag, 2000.

[38] F. Wang. Formal verification of timed systems: a survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, IEEE, 2004.

[39] G. Zhou, M.K. Leung, and E.A. Lee. A Code Generation Framework for Actor-Oriented Models with Partial Evaluation. In *International Conference on Embedded Software and Systems (ICESS)*, volume 4523 of *LNCS*, pages 786–799, Springer-Verlag, 2007.