Modeling State Diagrams with And-Cross Transitions

Opeyemi O. Adesina, Stéphane S. Somé, Timothy C. Lethbridge
School of Electrical Engineering and Computer Science
University of Ottawa, 800 King Edward Ave., Ottawa ON K1N 6N5, Canada
oadesina@uottawa.ca, ssome@eecs.uottawa.ca, tcl@eecs.uottawa.ca

***Abstract*—We propose an approach to encode state diagrams with and-cross transitions. The notion of and-cross transitions is being rejected by researchers and practitioners in model-driven engineering community for reasons that include limited use-in-practice, unmanageable underlying complexity, and availability of alternative modeling solutions. In this paper, we show that and-cross transitions can be useful in practice and its underlying structural complexity can be managed for the purpose of analysis.**

***Index Terms*—Formal Methods, Model-Driven Engineering, Umple, Symbolic Model Verification, State Machine Diagrams, And-Cross, Region-Cross.**

I. INTRODUCTION

We present a novel approach to modeling systems under analysis (SUAs) with and-cross transitions for the purpose of formal analysis. By and-cross transitions, we mean transitions whose sources and destinations states are located in parallel regions of an orthogonal state. For example, transition t_2 from *Emergency* to *Applied* (in Figure 1) is an and-cross transition. An orthogonal state is a composite state with regions whose submachines execute in parallel. The proposed approach is not limited by depth of and-cross transitions.

The underlying complexities of formal specification, verification and validation of safety-critical and embedded systems are increasing relentlessly [1]. Research efforts on managing these complexities have given rise to a variety of implementation solutions. These include solutions based on programming [2]–[4], modeling [4]–[6] and simulation of software abstractions [7]–[9] prior to implementation or deployment.

Model-Driven Engineering (MDE) [10] emerged as a disciplined approach to address software complexity and effectively represent domain concepts. It provides a level of abstraction demanded to represent components of software systems. MDE advocates domain-specific modeling languages, transformation engines and generators as means of managing emerging complexities in the software industry.

Umple is a model-driven engineering (MDE) technology that merges programming with modeling to facilitate the development and generation of complete software systems. In particular, it supports the model-code duality principle by representing models not only as diagrams but also text [4]. It provides succinct constructs for the representation of both static and dynamic aspects of software abstractions. These include class, state machine, and composite structure models.

In this work, our goal is to present an approach to compute enabling or disabling transitions for states and sub-state machines of an SUA. This cleanly separates concerns and eases analysis even in the presence of and-cross transitions irrespective of the depth. Our approach works on full Umple state machines, including those with deep nesting, and concurrent regions. In particular, we support and-cross (including what Faghih and Day called unusual transitions in [11]).

Harel's state-chart semantics (see [12], [13]) for and-cross transitions facilitate re-initializing every concurrent sub-machines (of an orthogonal state) but setting the target machine (i.e., host machine of the next state of and-cross transition) to the next state of the transition. The notion of and-cross is however being rejected by researchers and practitioners in model-driven engineering community for its limited use in practice; underlying complexity; and alternative modeling solutions. This is exemplified by the recent removal of and-cross transitions support from the UML [14].

However, since the goal of MDE is to provide sufficient level of abstraction to manage the complexities arising from the development of modern-day software systems, we deem it important to facilitate the representation and analysis of and-cross transitions. The construct is simple and sufficiently abstract to be substituted with the details provided by alternative solutions. Consequently, we focus on providing an approach to manage the complexities of and-cross transitions for the purpose of formal analysis by model checking.

In the literature, various encodings or implementations exist for the representation of state diagrams for symbolic model verification and the reasoning of temporal properties. These include RSML2SMV [15], BSML2SMV [11], SMUML [16] and STATEMATE2SMV [17]. But general problems with these tools or approaches include: *a complete neglect of or only partial solutions to and-cross*. We introduce modeling strategies to address these issues. First, we cleanly separate concerns but systematically integrate components of hierarchical systems. We also assume deterministic transitions during the transformation as our algorithm presented in [18] can be applied to compute a set of pairs of potentially conflicting (non-deterministic) transitions that can be further analyzed for actual cases of non-determinism.

The rest of the paper is organized as follows: In Section II, we present a modeling example that inspired this work. Section III presents formal background on the syntax and semantics of Umple. Section IV presents our approach to handling and-crossing by example. Section V discusses the state of research

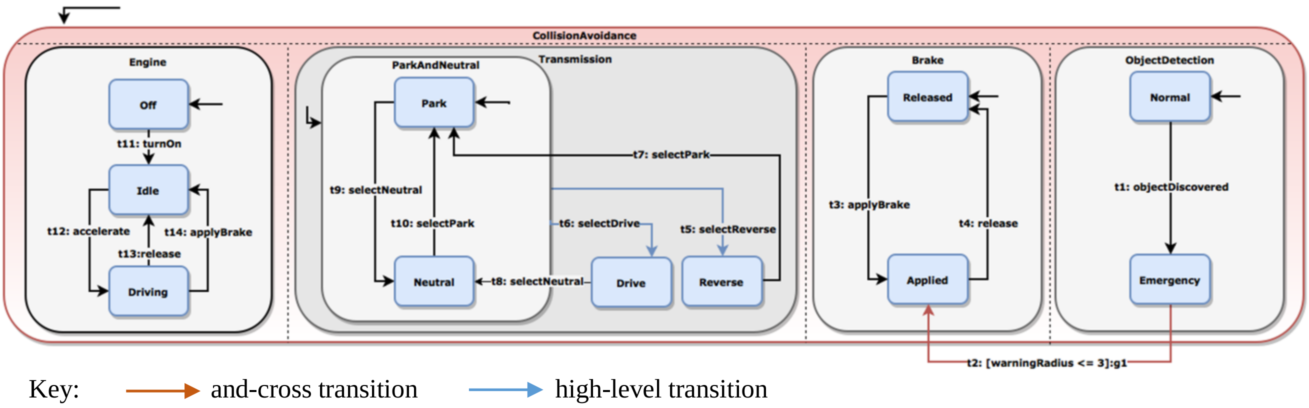


Figure 1. Visual Representation of a Simplified Collision Avoidance System

on and-crossing for model checking. Section VI concludes this work and state future thoughts.

II. Motivating Example

In this section, we present a modeling example with the goal of demonstrating the need (or an application) of *and-cross transitions* in the design of real-world systems. There may be a possibility of modeling the system we presented here with an alternative approach but we only assume that and-cross transition will facilitate a more intuitive representation of the expressed concept.

We consider a collision avoidance feature of a vehicle with the focus on the aspect that takes action autonomously without driver input (i.e., by braking at a lower speed). In particular, a typical use case relevant to our example may involve the activation of a collision detection system when a stationary object (e.g., vehicle, pedestrian, animal, etc.) is detected in a close-range (e.g., *warning radius – 3 meters*) and requires maximum braking force. Collisions in this category may be forward or rearward. For example, a realistic automotive case is experienced whenever a threat is detected during parallel parking. Avoiding a collision in this instance demands issuing a warning and actuating the braking mechanism autonomously.

Let us assume a collision system is a collection of parallel sub-systems such as *object detection*, *braking*, *transmission* and *engine*. The *object detection* sub-system is equipped to periodically send signals to its environment so as to detect objects within its range of detection. If an object is discovered, it is classified according to its position and heading and an input signal is generated from the environment indicating an object is discovered. The object detection sub-system is initially set to "*Normal*" but transitions to "*Emergency*" mode whenever an object is detected. At this point, the braking system is autonomously invoked.

The automatic braking sub-system (ABS) is initially "*Released*". Whenever the brake is applied, a signal is generated and the brake changes to "*Applied*" from its initial mode. Suppose the driver releases the brake, it is reset to the initial mode. The transmission sub-system can be in "*Park*", "*Neutral*", "*Reverse*" and "*Drive*" modes. We regard "Drive"

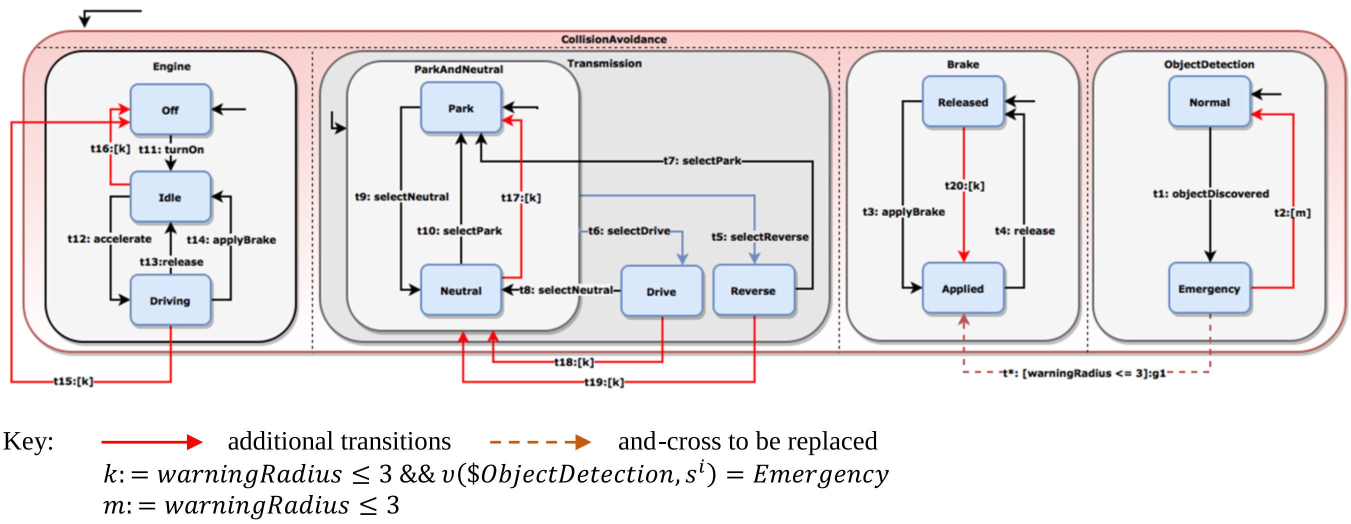


Figure 2. Alternative Modeling Solution to the Collision Avoidance System

as a black box entity in the design.

The engine sub-system can assume any of the following modes: "*Off*", "*Idle*", and "*Driving*". A release trigger causes a driving engine to become idle but accelerate causes a change of state to "*Driving*" and so on. The details of our motivating example are expressed graphically in Figure 1.

And-cross semantics is relevant in this case because in an emergency situation, *the brake is applied, object detection and transmission are reset to normal and park respectively*, and *engine is turned off* (to avoid potential risk of fire, etc.) in a step. In essence, all other parallel submachines of orthogonal state – "*CollisionAvoidance*" must be re-initialized except the brake whose state must be changed to a non-initial state - "*Applied*". As an alternative solution to the use of *and-cross transition*, this semantics may be modeled by introducing several transitions from non-initial states (e.g., "*Idle*") to the initial states of their parent machine (for non-targeted machines) such that whenever *emergency* arises, the non-targeted submachines are reset to their initial states. We present this modeling approach in Figure 2.

To facilitate comprehension, we introduce operator "$" such that given a non-orthogonal composite state, say A then "$A" is the corresponding state machine of "A". Similarly, we introduce environment operator "υ" such that $\upsilon(\$A, s^i)$ is the state of "$A" at step s^i.

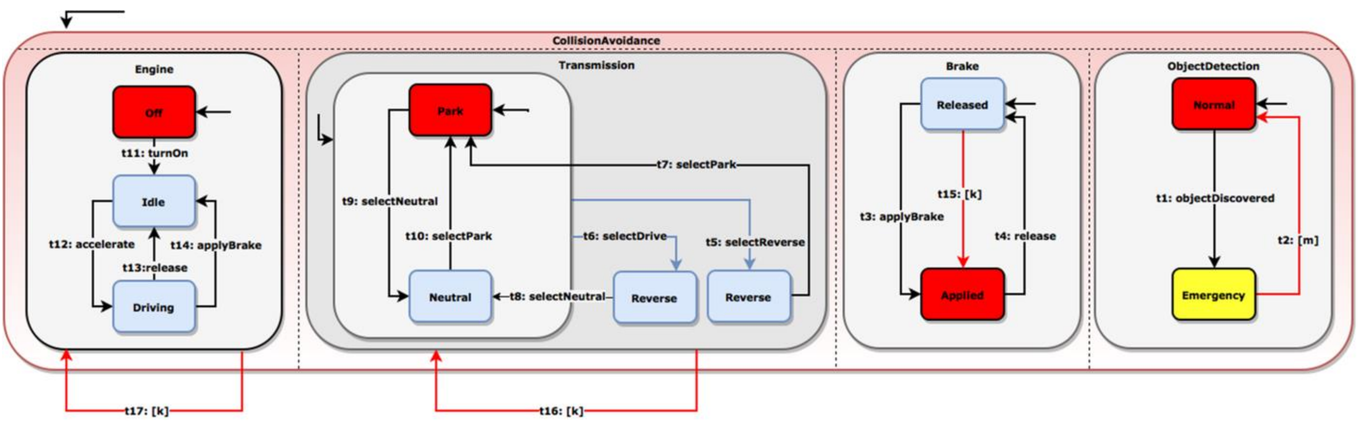
According to Figure 2, we would need additional 7 transitions to model the semantics under consideration as opposed to one (1) *and-cross transition*. These are represented in red arrows. They include "*Driving* → *Off*" (i.e., t_{15}), "*Idle* → *Off*" (i.e., t_{16}), "*Drive* → *ParkAndNeutral*" (i.e.,t_{17}), "*Reverse* → *ParkAndNeutral*" (i.e.,t_{19}), "*Drive* → *ParkAndNeutral*" (i.e.,t_{18}), "*Emergency* → *Normal*" (now t_2) and "*Released* → *Applied*" (i.e., t_{20}). The execution of a subset of these transitions will reinitialize all the submachines except the Brake (i.e., set to "*Applied*"). We require these transitions to ensure there are paths to the initial states of these submachines at any given time irrespective of the current states.

Listing 1. Collision Avoidance System (Umple)

```
class Example {
 Integer warningRadius;
 sm {
  CollisionAvoidance {
   ObjectDetection {
    Normal { objectDiscovered -> Emergency; } //t1
    Emergency { [warningRadius <= 3] -> Applied; }
     //t2
   }
   ||
   Brake {
    Released { applyBrake -> Applied; } //t3
    Applied { release -> Released; } //t4
   }
   ||
   Transmission {
    ParkAndNeutral {
     selectReverse -> Reverse; //t5
     selectDrive -> Drive; //t6
     Park { selectneutral -> Neutral; } //t9
     Neutral { selectPark -> Park; } //t10
    }
    Reverse { selectPark -> Park; } //t7
    Drive { selectNeutral -> Neutral; } //t8
   }
   ||
   Engine {
    Off { turnOn -> Idle; } //t11
    Idle { accelerate -> Driving; } //t12
    Driving {
     release -> Idle; //t13
     applyBrake -> Idle; //t14
   }}}}}
```

In Figure 3, we present another alternative solution to the same problem. The figure represents the maximum possible abstraction without the use of and-cross transitions. It requires four additional transitions as opposed to seven transitions in Figure 2. These include "*Emergency* → *Normal*" (i.e., t_2), "*Released* → *Applied*" (i.e., t_{15}), "*Transmission* → *Transmission*" (i.e., t_{16}), and "*Engine* → *Engine*" (i.e., t_{17}).

However, a single *and-cross transition* (i.e., t_*) is required with the semantics of such transition being sufficient for the required initialisations based on Harel's state machine semantics. In particular, suppose t_* is enabled at step s^{j-1} and



Key: additional transitions

$k := warningRadius \leq 3 \; \&\& \; \upsilon(\$ObjectDetection, s^i) = Emergency$

$m := warningRadius \leq 3$

Figure 3. Alternative Modeling Solution to the Collision Avoidance System (Maximum Abstraction)

executes at step s^j then

$\upsilon(\$\text{Brake}, s^j) = Applied$,

$\upsilon(\$\text{Transmission}, s^j) = ParkAndNeutral$,

$\upsilon(\$\text{ParkAndNeutral}, s^j) = Park$,

$\upsilon(\$\text{ObjectDetection}, s^j) = Normal$, and

$\upsilon(\text{Engine}, s^j) = Off$.

This is synonymous to what is achievable with the 7 transitions of the alternative solution. We can thus observe better abstraction as advocated by MDE with and-cross transitions. Therefore, managing the underlying complexities of the notion of and-crossing becomes critical and the focus of this paper.

III. Background

We present the syntax and semantics of Umple. To ease readability of Listing 1 simple state names are green; composite state names are red; and keywords are blue.

Umple is a model-oriented programming technology for the development of real-world software systems. It supports the model-code duality principle by representing software models interchangeably as diagrams and text [4]. Umple allows developers to model static views (i.e., class diagrams) and dynamic views (i.e., state diagrams) of software systems and automatically generates code for languages such as Java, C++, Ruby, and Php from the model. Umple achieves this by providing constructs and environments to express an extended subset of the Unified Modeling Language (UML) [19]. By extended, we mean that we provide a few features that UML does not support; for example, UML 2.5 [14] does not allow and-cross (or region-cross) transitions but we support this notion. Similarly, Umple allows developers to merge modeling with programming [20]. In particular, algorithmic logics are expressible in the dialect of the target language (e.g. Java, C++, Ruby, Php, etc.).

Listing 1 is an Umple version of the collision avoidance system presented in Figure 1. We will use this to describe the constructs of Umple.

A. Informal Syntax and Semantics of Umple

The notion of state machines as facilitated by Umple provides constructs to represent states (simple, composite, and orthogonal), transitions (regular, guarded, high-level), and events. State machines in Umple can either be simple or hierarchical. A simple state machine is composed of a set of simple states. For a hierarchical state machine, there are one or more composite or orthogonal sub-states.

An example of a simple state is "*Emergency*" defined in line 7 of Listing 1; an example composite state is "*ParkAndNeutral*" defined in lines 17-25 of Listing 1.

Orthogonal states (e.g., "*CollisionAvoidance*" defined on lines 4-33 of Listing 1) exist when two or more sub-states all become activated whenever control is transferred to their parent. For example, sub-states "*ObjectDetection*", "Brake", "*Transmission*" and "*Engine*" become activated whenever "*CollisionAvoidance*" is activated. In other words, the execution of sub-states occurs concurrently. Child states of composite and orthogonal states may be simple, composite, or orthogonal themselves.

By default, the first state (i.e., simple or composite) defined within an Umple state machine at any level of the hierarchy is regarded as its initial state (e.g., the state "*CollisionAvoidance*" in Listing 1).

Associated with a state, there can be any number of transitions. A state with zero outgoing transition is considered as an "end" state. A state can also be tagged as 'final' implying the end of the object's life whenever control is transferred to that state. Several transitions, indicated by "->" are shown in Listing 1. The state enclosing the transition is referred to as the 'source'; while the target state is referred to as the 'destination'. For transition t_4 on line 13 of Listing 1, "Applied" and "Released" are its source and destination states respectively. A basic transition is a transition without a guard statement.

A guarded transition has a Boolean expression controlling whether or not a transition is taken whenever an event arrives or automatically taken whenever it is an auto-transition. A high-level transition is any kind of transition defined outside of sub-states in a composite state, but which has effect in all the sub-states (e.g., t_5, line 18 of Listing 1). Associated with every transition is an event (e.g., "*selectPark*" on line 23 of Listing 1).

Although, the subset of Umple considered in this work excludes actions, we consider it necessary to discuss variables and types supported in Umple since the values of guards are computed from the variables at every step of execution. Umple allows the use of primitive and non-primitive data types. The primitive data types of focus are *Boolean*, *Integer* and *Real*. For example, on line 2 of Listing 1 we present a definition of an Integer variable "*warningRadius*" defaulted to "0".

Umple transforms hierarchical state machine (e.g., "sm" of Listing 1) internally into a collection of state machines. For each non-orthogonal composite state, there is a corresponding state machine such that its sub-states become the states of the state machine when active. We introduce a special 'null' state for every state machine such that these state machines are in their 'null' state until they are being activated. Similarly, for every region of orthogonal states, there is also a corresponding state machine. A state machine is also generated for the root state machine but without the 'null' state, since the root is always active throughout its containing object's life cycle. But sub-state machines are only active whenever control is transferred to their parent state or any of their sub-states [4], [21].

B. Formal Syntax and Semantics of Umple

To facilitate easy discussion of concepts involved in our approach to managing and-crossing, a formal description of syntax and semantics of Umple becomes critical.

A state machine in Umple (e.g., "sm", lines 4-35), say A, is a 6-tuple $\langle n_A, S_A, l_A, s_A^0, V_A, R_A \rangle$; such that n_A is the name, S_A is a finite set of top-level states (excluding indirect sub-states), $s_A^0 \in S_A$ is the initial state (e.g., "*CollisionAvoidance*"), l_A is a finite set of labels, V_A is a finite set of variables (or attributes –

"*warningRadius*") and $R_A \subseteq S_A \times l_A \times S_A$ defines a transition relation on A. By 'indirect sub-state' of a state machine, we mean a state whose ancestor is a top-level state of the state machine under consideration irrespective of its depth. A submachine of a composite state is a state machine encapsulating a region enclosed by the state. For example, a non-orthogonal composite state encloses a region but an orthogonal composite state encloses at least two regions. In our approach a state machine is defined for every region.

Let U_s be the universal set of states of a SUA. U_s contains all the states of the model at any level (e.g., *Engine*, *Idle*, *Park*, *Reverse*, *Brake*, *Emergency* etc.). Let U_l (e.g., $\{t_i | 1 \leq i \leq 14\}$ for Figure 1) be the set of all labels in the SUA and R be the universal set of transitions of the SUA such that $R \subseteq U_S \times U_l \times U_S$.

Let V_A be a set of pairs $\langle n_v, t_v \rangle$, where n_v is a name and $t_v \in \mathbb{T}$ is a type of a given variable $v \in V_A$, such that $\mathbb{T} = \{\text{integer}, \text{boolean}, \text{real}\}$. We consider G_l as the universal set of guards, E_l as the universal set of non-parameterized events, and A_l as the universal set of actions such that $l_A \subseteq G_l \times E_l \times A_l$..

Let $\gamma: l_A \to E_l$ such that $\gamma(g, e, a) = e$ We also define $L: R_A \to l_A$ such that $L(s, l, s') = l$.

Given any state $s \in S_A$, we define a mapping function $\beta: s \to \mathbb{N}$ to map a state, s to its number of sub machines; such that $\forall_{s \in S_A} \beta(s) = 0, 1, n$ when s is simple, non-orthogonal composite and orthogonal composite states respectively where $n > 1$. For example, $\beta(\text{Applied}) = 0$, $\beta(\text{ParkAndNeutral}) = 1$, and $\beta(\text{CollisionAvoidance}) = 4$.

We say M is a simple state machine if and only if $\forall_{s \in S_M}$ $\beta(s) = 0$ and it is hierarchical $\exists_{s \in S_M}$, such that $\beta(s) \geq 1$. To facilitate the specification of hierarchical structures, we introduce a binary relation $\sqsubseteq$ on W, such that W is the set of all sub-state machines (including the root). If $M, L, Z \in W$ then $M \sqsubseteq L$ specifies L as a direct ancestor or a parent state machine of M. $\ddot{\sqsubseteq}$ is the reflexive closure of $\sqsubseteq$ on W. Where $M \ddot{\sqsubseteq} Z$ defines M as a submachine of Z, such that Z is ancestrally-related to M. We introduce a partial mapping function $\rho : M \to U_S$ such that $\rho(M) = s$, s being the parent state of M. Hence, we say Z is a root state machine iff $\rho(Z)$ is *undefined*. On the other hand, for any non-root state machine M, $\rho(M)$ is defined. In particular, there can only be a submachine M if and only if S_L being a set of states of state machine L, $\exists_{k \in S_L}$, $\rho(M) = k$ and $M \sqsubseteq L$.

Recall that k is an orthogonal state iff $\beta(k) \geq 2$. In essence, there are at least two sub-state machines for k. We introduce the relation $\sqsubseteq_{||}$ to define parallelism between any two orthogonal state machines, such that $M \sqsubseteq_{||} L$ iff $\exists_{y \in S_H}$, $\rho(M) = \rho(L) = y$ and $M \sqsubseteq H, L \sqsubseteq H$.

A pair of elements $j, s \in U_s$ such that $j \cong s$, implies s (e.g., Engine, line 27-33) is ancestrally related to j (e.g., Idle, line 29. Let U_t be a universal set of transitions of the SUA (e.g., $\{t_i | 1 \leq i \leq 14\}$) such that if Z is a root state machine, then $U_t = R_Z$. Let $F, X \subseteq U_t \times 2^{U_s}$ such that $t \in U_t$, $(t, F_s)\ in\ F$ and $(t, X_s)\ in\ X$ where $F_s, X_s \in 2^{U_s}$ are sets of *from (source) and next (destination) states* of transition t respectively.

IV. Our Work

In this section, we present our approach to enabling and disabling states and sub-machines for and-cross transitions. However, we present more formal definitions of some concepts to facilitate the discussion.

Definition 1. And-Cross Transition

Given an orthogonal composite state, say k (i.e., $\beta(k) \geq 2$) and M, N are two parallel sub-state machines of k ($\rho(M) = \rho(N) = k$ and $M \sqsubseteq_{||} N$); O, P such that $O \ddot{\sqsubseteq} M \wedge P \ddot{\sqsubseteq} N$; s_1, s_2 embedded states such that $s_1 \in S_O, s_2 \in S_P$; any transition t such that there are $(t, F_s)\ in\ F, (t, X_s)\ in\ X$ with $s_1\ in\ F_s$ and $s_2\ in\ X_s$ is an and-cross transition. The set of and-cross transition of a composite state k is denoted as $\delta(k)$.

We introduce an operator $\varepsilon(M)$ which defines a union of a set of transitions whose target state is external to s, an orthogonal but parent state of M and a set of external-and-cross transitions of s with respect to M.

Definition 2. External Transition

We say transition t is an "external transition" of a composite state s, if there is a state k such that $k \cong s$ and sub-state machines M, N such that $\rho(s) = M, M \sqsubseteq N$ then $t \in R_N$ is external with respect to s whenever $k \notin X(t)$. We represent this transition set as $\text{ext}(s)$.

Definition 3. External-And-Cross Transition

An external-and-cross transition t of state s with respect to sub-machine M (denoted as $\omega(s, M)$), where $t \in \delta(s)$ and $s = \rho(M)$. If there exist states j, x such that $j \in S_M$, $x \ncong j$ and $x \cong s$. If $x \in X(t)$ then t is an external-and-cross transition.

Hence, $\varepsilon(M)$ is simplified as $\{t, t'\}$ where $t \in \omega(s, M), t' \in ext(s)$.

Definition 4. Disabling Transition

Transition t disables sub-machine M if there exists a state s such that $s \in S_N$, $t \in R_N$, $M \sqsubseteq N$, $s = \rho(M)$ and $t \in \varepsilon(M)$. In other words, a sub-machine is disabled or will remain disabled whenever a transition whose next state is the parent state or non-embedded sub-state of the parent state of the sub-machine under consideration execute (i.e., external transition) or an and-cross transition whose destination state is not a direct or indirect sub-state of the sub-machine is fired. By "non-embedded sub-state" of a composite state s, we mean states outside the boundaries of s but local to the host state machine

Listing 2. $ParkAndNeutral sub-machine (SMV)

```
MODULE $ParkAndNeutral ( sm, transmission )
 VAR
  state : { Park , Neutral , null };
 ASSIGN
  init( state ) := null;
  next( state ) := case
   sm.t5 | sm.t6 : null; -- disabling transitions
   sm.t10 | sm.t7 : Park; -- enabling transitions
   sm.t9 | sm.t8 : Neutral; -- enabling transitions
   transmission.state = ParkAndNeutral &
     state = null : Park;
   TRUE : state;
  esac;
```

of s.

By this definition, not only orthogonal submachines are disabled but also non-orthogonal ones. For example, in Figure 1 "*ParkAndNeutral*" is a non-orthogonal composite state and "*ParkAndNeutral*" is its sub-state machine. The SMV representation of "$ParkAndNeutral" sub-state machine is presented in Listing 2. Its set of disabling transitions is given as: $\{t_5, t_6\}$. We computed this set as follow:

a) "*$ParkAndNeutral*" $\sqsubseteq$ "*$Transmission*".
b) $t_{\$Transmission} = \{t_i \mid 5 \leq i \leq 10\}$.
c) ω(*ParkAndNeutral*, *$ParkandNeutral*) = $\emptyset$.
d) *ext*(*ParkAndNeutral*) = $\{t_6, t_5\}$.
e) $\varepsilon(M) = \{t_5, t_6\} \cup \emptyset = \{t_5, t_6\}$.

On another hand, the set of disabling transitions for an orthogonal submachine is dependent on the destination or source of an and-cross transition. In particular, an orthogonal sub-machine whose sub-state is a destination of an and-cross transition may be disabled only by external transitions of its parent state. For example, the set of disabling transition for "*$Brake*" is $\emptyset$. We determined this as follow:

a) t_2 is an and-cross transition from "*$ObjectDetection*" to "*$Brake*".
b) "*$Brake*" $\sqsubseteq$ "*$Sm*".
c) $t_{\$Sm} = \{t_i \mid 1 \leq i \leq 10\}$.
d) ω(*CollisionAvoidance*, *$Brake*) = $\emptyset$.
e) *ext*(*CollisionAvoidance*) = $\emptyset$.
f) $\varepsilon(M) = \emptyset \cup \emptyset = \emptyset$.

Listing 3. $Brake sub-machine (SMV)

```
MODULE $Brake ( sm )
 VAR
  state : { Released , Applied , null };
 ASSIGN
  init( state ) := null;
  next( state ) := case
   sm.t4 : Released; -- enabling transitions
   sm.t2 | sm.t3 : Applied; -- enabling transitions
   sm.state = CollisionAvoidance &
    state = null : Released; -- default activation
   TRUE : state;
  esac;
```

The SMV representation of the "$Brake" sub-state machine is presented in Listing 3. Since the disabling set is empty, there was no expression assigning "null" to the state variable of the sub-state machine.

An orthogonal sub-machine whose sub-state is not a destination of an and-cross transition may be disabled by at least and-cross transitions whenever the set of and-cross transition of its parent state is non-empty. Let us consider "*$Engine*" sub-state machine. The set of disabling transition is given as: $\{t_2\}$. We derived this as follow:

a) t_2 is an and-cross transition from "*$ObjectDetection*" to "*$Brake*".
b) "*$Engine*" $\sqsubseteq$ "*$CollisionAvoidance*".
c) $t_{\$CollisionAvoidance} = \{t_i \mid 1 \leq i \leq 10\}$.
d) ω(*CollisionAvoidance*, *$Engine*) = $\{t_2\}$.
e) *ext*(*CollisionAvoidance*) = $\emptyset$.
f) $\varepsilon(M) = \{t_2\} \cup \emptyset = \{t_2\}$.

Listing 4. $Engine sub-machine (SMV)

```
MODULE $Engine ( sm )
 VAR
  state : { Off , Idle , Driving , null };
 ASSIGN
  init( state ) := null;
  next( state ) := case
   sm.t2 : null; -- disabling transitions
   sm.t13 | sm.t11 | sm.t14 : Idle;
   sm.t12 : Driving; -- enabling transitions
   sm.state = CollisionAvoidance &
    state = null : Off; -- default activation
   TRUE : state;
  esac;
```

The SMV representation of the "$Engine" sub-state machine is presented in Listing 4. Note that whenever a member of the set of transitions external to the parent state of a sub-state machine under consideration executes, it is guaranteed that the machine is inactive at any given step. The disabling transitions of a machine are therefore necessarily a subset of transitions of the state machine containing its parent state.

The process of enabling a sub-state machine differs. In particular, we enable a machine based on the transitions enabling its direct or indirect sub-state(s) or by default. By "default", we mean enabling the machine whenever a transition whose next state includes the parent state of the machine under consideration executes. We introduce operator $\nu : M \rightarrow S_M$ to facilitate further discussion, such that, $\nu(M)$ is the current state of the machine M. We formally define state activation by default as follows:

Definition 5. Activation by Default

State $s \in U_s$ is activated by default in a micro-step, say s'' if and only if there exist module M, state j such that $s = s_M^0$, $j = \rho(M)$, $\nu(M) = null$ and j becomes enabled at the previous micro-step s'.

For example, "*$ObjectDetection*" sub-state machine will be enabled whenever state "Normal" is activated by default. Furthermore, we define a transition set to enable states of a state machine.

Definition 6. Enabling Transition

Transition $t \in U_l$ is an enabling transition of state s if and only if $s \in X(t)$ or there is a state $k \in U_s$ such that $k \cong s$ and $k \in X(t)$. In other words, a state is enabled by a set of transitions into itself and those of its sub-states.

For example, consider simple state "Applied" of Figure 1, the set of transitions enabling this state are t_2. The set of enabling transitions of non-orthogonal composite state "*ParkAndNeutral*" are: $\{t_i \mid 5 \leq i \leq 10\}$. To enable orthogonal composite state "*CollisionAvoidance*" the following set of transitions are relevant: $\{t_i \mid 1 \leq i \leq 10\}$

For a concurrent machine, whenever a transition into a sub-state of a parallel machine executes, the parent state is activated in the same micro-step and the parallel machines are activated in the next micro-step. These cycle continues until the system assumes a stable state. In other words, the semantics we adopted for this case is "*run-to-completion*" paradigm [14]. A global configuration based on our approach is defined as follows:

DEFINITION 7. GLOBAL CONFIGURATION

A global configuration of an Umple state machine A is a quadruple $\langle M \times U_s \times E_s \times V_A \rangle$ such that M is the set of sub-state machines (including the root), E_s is a set of execution steps and V_A is a finite set of pairs $\langle n_a, v_a \rangle$ such that n_a is a variable name and v_a is its value. Equation (1) defines the configuration of an SUA at step i where sub-machine m_1 is in state s_1, variable n_1 is evaluated to value v_1, and k, j are the number of variables and sub-machines respectively.

$$\langle \langle m_1, \ldots, m_j \rangle, \langle s_1, \ldots, s_j \rangle, s^i, \langle \langle n_1, \ldots, n_k \rangle, \langle v_1, \ldots, v_k \rangle \rangle \rangle \quad (1)$$

If an enabled transition fires, it may lead to a change in configuration. In particular, it may change the states of sub-machines (for non-self-transitions) as well as the values of variables (for transitions with non-empty actions statements). In summary, a sub-state machine becomes active in a micro-step its sub-state is activated. This includes the micro-step at which its initial state receives control or activated by default. However, it is worthy to note that the root state machine is enabled throughout the life cycle of the model under analysis. We will also like to emphasize that our approach is not limited by the depth of and-cross transitions. The set of transitions composed for enabling and disabling states and sub-state machines as presented in this paper have been implemented in Umple2SMV [22].

V. RELATED WORK

And-cross transition has gained limited attention in model-driven engineering community due to its complexity or availability of alternative solutions or their limited use-in-practice. We present an overview of the literature to discuss the state of research on and-crossing.

Faghih and Day [11] propose an algorithm for creating a semantics-based, parameterized translator from a family of big-step modeling languages (BSML) to SMV for model checking purposes. The translator developed allowed configuration of semantics for different modeling languages. It handles various kinds of consistency checks (including non-determinism) and allows the specifications of and-cross transitions. However, and-crossing within a child and-state of an and-state is not supported. This kind of transitions is what Faghih and Day referred to as unusual transition. The semantics of and-cross transitions are based on the concept of scope. A scope defines the least-common ancestor to the source and destinations of a transition. Hence, the sub-states to be exited and entered are determined.

Harel et. al [13] presents Rhapsody, an implementation of the semantics of state charts [12]. Rhapsody allows the execution of object-oriented state charts. Among various notions facilitated are And-state, Or-state and basic states, join, connector, etc. It also facilitates the representation of compound transition (including and-crossing). The semantics of compound transitions are based on the concept of scope as in [11].

Chan et al. [23] translated RSML specification of TCAS II system to SMV for symbolic model verification. RSML (Requirements State Machine Language) is a communicating state machine model. It includes features like parallel state machines and hierarchical abstractions. Composite structures are abstracted into super-states. The applicability of the approach presented to and-crossing is not discussed.

Sreemani and Atlee [24] implemented a program that translates a requirement expressed in Software Cost Reduction (SCR) into an equivalent SMV specification. The focus of their work is to demonstrate the feasibility of symbolic model checking to reason about temporal properties of large software specifications. They successfully applied their approach and falsified some properties. However, the work lacks details on the notions facilitated and and-crossing is also not discussed.

VI. CONCLUSION AND FUTURE WORK

We presented an approach to model the semantics of and-cross transitions for the purpose of symbolic model verification. The need for and-crossing was exemplified in this work. We also present both informal and formal background on Umple to facilitate formal discussion of our approach to enable or disable sub-machines and states involved in and-crossing. We applied our approach to the motivating example to demonstrate how these sets are being composed.

Our long-term goal is to apply this technique to several industrial case studies for the formal verification of complex systems modeled using and-cross transitions.

VII. REFERENCES

[1] O. Adesina, "Integrating formal methods with model-driven engineering," in *18th International Conference on Model Driven Engineering and Languages and Systems (MODELS'15)*, 2015, pp. 6–10.

[2] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. H??hnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich, "The KeY platform for verification and analysis of Java programs," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8471, pp. 55–71.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, vol. 36, no. 5, pp. 203–213.

[4] O. Badreddin and T. C. Lethbridge, "A manifestation of model-code duality: facilitating the representation of state machines in the umple model-oriented programming language," 2012.

[5] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming," *ASE'07 - 2007 ACM/IEEE Int. Conf. Autom. Softw. Eng.*, pp. 547–548, 2007.

[6] B. Bordbar and K. Anastasakis, "UML2ALLOY: A tool for lightweight modelling of discrete event systems.," *Guimarães, N., Isaías, P. IADIS Int. Conf. Appl. Comput. 2005*, no. 1999, pp. 209–216, 2005.

[7] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 256–290, 2002.

[8] A. Cimatti, E. Clarke, and E. Giunchiglia, "Nusmv 2: An opensource tool for symbolic model checking," in *International Conference on Computer Aided VerificationComputer Aided Verification*, 2002, vol. 2404, pp. 359–364.

[9] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The NUXMV Symbolic Model Checker," in *26th International Conference on Computer Aided Verification*, 2014, pp. 334–342.

[10] D. C. Schmidt, "Model-Driven Engineering," *Comput. Comput. Soc.*, no. February, pp. 25–31, 2006.

[11] F. Faghih and N. A. Day, "Mapping Big-Step Modeling Languages

to SMV," 2011.

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[13] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)," *Integr. Softw. Specif. Tech. Appl. Eng.*, vol. 3147, pp. 325–354, 2004.

[14] Omg, "OMG Unified Modeling Language: Version 2.5," vol. 5, no. September, p. 786, 2013.

[15] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese, "Model checking large software specifications," *IEEE Trans. Softw. Eng.*, vol. 24, no. 7, pp. 498–520, 1998.

[16] J. Dubrovin and T. Junttila, "Symbolic model checking of hierarchical UML state machines," in *8th International Conference on Application of Concurrency to System Design*, 2008, pp. 108–117.

[17] E. M. Clarke and W. Heinle, "Modular translation of statecharts to SMV," 2000.

[18] O. O. Adesina, T. C. Lethbridge, and S. S. Somé, "A Fully Automated Approach to Discovering Nondeterminism in State Machine Diagrams," in *10th International Conference on the Quality of Information and Communications Technology*, 2016, pp. 73–78.

[19] J. Rambaugh, I. Jacobson, and G. Booch, *Advanced Praise for The Unified Modeling Language Reference Manual , Second Edition*, 2nd ed. Toronto: Addison-Wesley, 2004.

[20] T. C. Lethbridge, V. Abdelzad, M. H. Orabi, A. H. Orabi, and O. O. Adesina, "Merging Modeling and Programming Using Umple," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, vol. 17, T. Margaria and B. Steffen, Eds. Corfu: Springer International Publishing, 2016, pp. 187–197.

[21] O. Badreddin, T. C. Lethbridge, A. Forward, M. Elasaar, and H. Aljamaan, "Enhanced Code Generation from UML Composite State Machines," in *Modelsward 2014*, 2014.

[22] CRUiSE, "UmpleOnline: Generate Java, C++, PHP, or Ruby from Umple code," *UmpleOnline: Generate Java, C++, PHP, or Ruby from Umple code*, 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umpleonline/. [Accessed: 18-Jan-2016].

[23] W. Chan, S. Member, R. J. Anderson, P. Beame, S. Burns, D. Notkin, S. Member, and J. D. Reese, "Model Checking Large Software Specifications," *IEEE Trans. Softw. Eng.*, vol. 24, no. 7, pp. 498–520, 1998.

[24] T. Sreemani and J. M. Atlee, "Feasibility of Model Checking Software equirements : A Case Study," in *COMPASS '96, Systems Integrity. Software Safety. Process Security.*, 1996, pp. 77–88.