UCL DEPARTMENT OF COMPUTER SCIENCE

**Research Note**
RN/12/02

# Dependence for Slicing State-based Models: A Survey

2 April 2012

***Kelly Androutsopoulos***

.

## Abstract

Dependence is a relation that determines which parts of a system influence the computation of another part. Dependence analysis underpins many activities in computer science, such as model checking, debugging, slicing, security. We are interested in dependence analysis for slicing for finite state machine-based models. In this paper we survey existing dependence relations defined for slicing at the model-level as there has been recently much interest from the slicing community.

# Dependence for Slicing State-based Models: A Survey

Kelly Androutsopoulos

**Abstract**

Dependence is a relation that determines which parts of a system influence the computation of another part. Dependence analysis underpins many activities in computer science, such as model checking, debugging, slicing, security. We are interested in dependence analysis for slicing for finite state machine-based models. In this paper we survey existing dependence relations defined for slicing at the model-level as there has been recently much interest from the slicing community.

We survey definitions of dependence, in particular data, control, interference and any other dependence relations that are defined by finite state machine (FSM) slicing approaches. We only consider graphical FSM notations.

## 1 Data Dependence for Slicing FSMs

There have been two general approaches to defining data dependence. The first is based on the idea that an element $x$ is required to evaluate $y$. For example, a variable $y$ is defined in terms of $x$. This is not limited to variables, but can be applied to other elements. For example, to execute a transition, a trigger event, source state and all ancestor states are required. We call this approach the *uses* approach from the uses relation defined by [HTW98] (see Definition 1).

The second is based on definition-clear paths of variables, i.e. a variable $v$ is defined in an element $x$ and used in an element $y$ and there exists a path from $x$ to $y$ where $v$ is not modified. We call this approach *definition-clear paths*. These types of definitions are given at different levels of granularity which could lead to more precise slices. For example, [Oja07] define data dependence between parts of transitions, rather than transitions, and slicing can remove these parts i.e. trigger events, guards or actions. We further divide these definitions according to whether they apply within an automaton or state machine (i.e. intra-automaton) or between parallel automaton or state machines (i.e. inter-automaton).

Table 1 groups the key papers on FSM slicing that define data dependence according to the classification that we have described. In the rest of this section we describe each data dependence definition in turn.

Table 1: A classification of key papers that define data dependence for FSM slicing.

| **Uses** | **Definition-Clear Paths** | |
|---|---|---|
| [HTW98] | **Intra-Automaton** | **Inter-Automaton** |
| [CABN98] | [KSTV03] | [WDQ02] |
| [SPH08] | [Oja07] | [Lan06] |
| [FL05] | [LG08] | Janowska and |
| | [ACH+09] | Janowski [JJ06] |

## 1.1 Uses Approach

The authors in [HTW98] have defined data dependence for RSML specifications as the set of elements required to determine the value of a particular variable, transition, function and macros (expressions in guards are defined as mathematical functions and macros are defined for frequently used conditions). Data dependence for variables, macros, and functions is simple as it uses the elements that are visible in the definition. Data dependence for transitions requires that transitions are dependent on their guarding conditions, their source state, and all ancestors of the source state (because of state hierarchy).

**Definition 1 (Data Dependence for RSML)** *Let $A$ be the union of sets of states, transitions, variables, constants, functions and macros in an RSML specification. [HTW98] define data dependence by the relation uses, which is a mapping $A \mapsto A$ where $uses(x, y)$ means that $y$ is required to evaluate $x$.*

The authors in [FL05] and [CABN98] have defined a general notion of dependence that is similar to Definition 1. In addition, [FL05] state that a target state and an action of a transition $t$ depend on its source state, triggering event and guard.

According to [SPH08], a variable in an assignment is data dependent on variables in the assigned expression.

## 1.2 Definition-Clear Paths Approach: Intra-Automaton

The authors in [KSTV03] have defined data dependence for EFSMs between variables on transitions. In particular, it is defined as a definition-clear path between a variable's definition at a transition $t_1$ and its use at transition $t_2$. This definition is also adopted by [ACH+09].

**Definition 2 (Data Dependence for EFSMs)** *Transition $t_2$ data depends [KSTV03] on transition $t_1$ with respect to variable $v$ if:*

1. *$v \in D(t_1)$, where $D(t_1)$ is a set of variables defined by transition $t_1$, i.e. variables defined by actions and variables defined by the event of $t_1$ that are not redefined in any action of $t_1$;*

2. *$v \in U(t_2)$, where $U(t_1)$ is a set of variables used in a condition and actions of transition $t_1$;*

3. *there exists a path from $target(t_1)$ to $source(t_2)$ whereby $v$ is not modified.*

The authors in [LG08] have introduced a (intra-automata) data dependence definition for communicating automata (IOSTs). Similarly to Definition 2, they extend the definitions of *define* and *use* because variables can be used in transition guards and can be defined or used in actions (variable substitutions). Their extensions must also be able to handle communicating actions, which is not required for Definition 2. Definition 3 shows how $D(t)$ and $U(t)$ in Definition 2 is extended for communicating actions, i.e. what is required in addition.

**Definition 3 (Definition/Use for Communicating Automata)** *[LG08] Let $t$ be a transition and $v$ a variable. Then $v \in D(t)$ if an action ($c?v$) is performed that causes the system to wait on some channel $c$ for the reception of a value to be assigned to $v$. $v \in U(t)$ if an action ($c!v$) is performed that causes the system to emit a message having the argument $v$ on channel $c$.*

The data dependence definition defined by [LG08] is similar to Definition 2, with an additional condition (condition three in Definition 4) that is required because of the semantics of IOSTSs as it may be possible that $v$ is redefined at $t_2$ by a input action.

**Definition 4 (Data Dependence for Communicating Automata)** *A transition $t_2$ is data dependent [LG08] on a transition $t_1$ if there exists a variable $v$ such that:*

1. *$v \in D(t_1)$;*

2. *there exists a path $\pi$ from the $target(t_1)$ to the $source(t_2)$ where $v$ is not defined;*

3. *and one of the following is true: a) $v$ is used in the $guard(t_1)$; or b) $v$ is not defined at $action(t_1)$ and $v \in U(t)$ where $t \in \pi$.*

The authors in [Oja07] have defined data dependence for UML state machines between nodes in a CFG. The nodes of the CFG represent different parts of UML state machine transitions. Each transition can correspond to several nodes. Table 2 lists the different types of CFG nodes defined and what part of the transitions they represent. Also, the last column shows whether the nodes have a $D$ and $U$ set. $D$ is the set of variables that are defined by the part of the transition. $U$ is the set of variables that are used by the part of the transition. Ojala differentiates between variables that are defined when entering the state and those that are defined when exiting the state. $D$ and $U$ sets are used to define definition-clear paths between CFG nodes. This notion of data dependence is similar to Definition 10 as it applies to parts of transitions and also to concurrent state machines.

| **CFG nodes** | **Part of Transition** | $D/U$ **sets** |
|---|---|---|
| BRANCH | Triggers and guards | $D$ = union of $D$ sets of its elements. $U$ = union of $U$ sets of its elements. |
| SIMPLE | Actions | $D$, $U$ |
| SEND | Actions | $D$, $U$ |

| **Components of BRANCH node** | **Part of Transition** | $D/U$ **sets** |
|---|---|---|
| Element | Trigger and guard | $D$, $U$ |
| Parameter | Event parameter | $D$ |

Table 2: CFG nodes and what parts of a transition they represent. The sub-structure of the BRANCH node is given in a separate table. Note that the end node is of type BRANCH and has no sub-structure. $D$ is the set of variables that are defined by the part of the transition. $U$ is the set of variables that are used by the part of the transition.

**Definition 5 (Data Depedence for UML state machines)** *Paths are defined as a sequence $a_1, .., a_i$ where each $a_j \in \{a_1, .., a_i\}$ is either a SIMPLE, SEND or end node, or b.e where b is a BRANCH node and e its element. Data dependence [Oja07] is defined in terms of definition-clear paths. Definition-clear paths with respect to variable v are paths $p_1, p_2, ..., p_{(n-1)}, p_n$ where v is defined at $p_1$ (i.e. $v \in D(p_1)$), v is used at $p_n$ ($v \in U(p_n)$) and v is not defined in $p_2, .., p_{(n-1)}$. The following definition-clear paths with respect to a variable v are defined:*

- *between a SIMPLE node $s_1$ and a SIMPLE, SEND or end node $s_2$;*
- *between a SIMPLE node $s_1$ and an element e in a BRANCH node b;*
- *between a parameter p (which is in an element and BRANCH) and a SIMPLE, SEND or end node $s_2$;*
- *between a parameter p in an element e1 in a BRANCH b1 and an ELEMENT e2 in a BRANCH node b2;*

*Also, a parameter p is data dependent on a node q if q evaluates an expression whose value gets assigned to p when an event is received. Node q is one of the CFG nodes that are created when a generated event occurs (either in the same UML state machine or in another).*

## 1.3 Definition-Clear Paths Approach: Inter-Automaton

The authors in [WDQ02] have defined three data dependence definitions for extended hierarchical automata (EHA): one for sequential automaton (not concurrent) and two for concurrent automaton. Note that variables are updated on states, rather than transitions. Since EHA are concurrent and hierarchical, the

definition of $D$ and $U$ differs from Definition 2 and it applies to states rather than transitions. A variable $v$ is used at state $x$ (i.e. $v \in U(x)$) if $v$ is referenced in the actions of state $x$ or it is referenced in the actions of any sub-state or transitions in the sub-EHA of $x$ and can be defined and referenced outside the sub-EHA of $x$. The set of defined variables $D(x)$ at a state $x$ includes all variables that have been defined (have values assigned to them) in the actions in sub-EHA of $x$ and can be defined and referenced outside the sub-EHA of $x$. Internal variables of a state $x$ are local variables that can only be defined and used in sub-EHA of $x$. $U$ and $D$ can also apply to transitions.

**Definition 6 (Sequential Data Dependence for EHA)** *A state or transition $y$ in a sequential automaton $A$ that uses a variable $v$ (i.e. $v \in U(y)$) is sequential data dependent [WDQ02] on a state or transition $x$ in $A$ that defines $v$ (i.e. $v \in D(x)$) if there is path in $A$ from $x$ to $y$ where $v$ is not modified.*

Sequential data dependence is defined as a definition-clear path between states or transitions in the same sequential automaton. Parallel data dependence is defined as a definition-clear path between states and transitions in concurrent automata.

**Definition 7 (Parallel Data Dependence (PDD) for EHA)** *Let $A$ and $B$ be two different sequential automata, and $s_A$ is a state in $A$, $t_A$ is a transition in $A$, $s_B$ is a state in $B$ and $t_B$ is a transition in $B$. If $A$ and $B$ are sub-states of $C$ then $s_B$ is parallel data dependent [WDQ02] on $s_A$ ($s_A \xrightarrow{PDD} s_B$) iff $U(s_A) \cap D(s_B) \neq \emptyset$. Similarly $s_A \xrightarrow{PDD} t_B$, or $t_A \xrightarrow{PDD} s_B$, or $t_A \xrightarrow{PDD} t_B$ iff $U(s_A) \cap D(t_B) \neq \emptyset$, or $U(t_A) \cap D(s_B) \neq \emptyset$, or $U(t_A) \cap D(t_B) \neq \emptyset$ respectively.*

Refinement data dependence [WDQ02] is defined between states and transitions of concurrent automata. Some state $x2$ is refinement dependent on $x1$, where $x2$ is in a sub-sequential automaton of an element $x1$, if the value of some variable computed at $x1$ is the value that $x2$ will return, or some event generated in $x1$ is used to synchronise with some concurrent state of $x2$. It differs from parallel data dependence as it computes dependencies in sub-states rather than across concurrent states.

**Definition 8 (Refinement Data Dependence (RDD) for EHA)** *$GE(s)$ is the set of all events that are generated in the actions in sub-states of state $s$ and can be used as the trigger events of transitions outside of $s$. Similarly $GE(t)$, for a transition $t$, is the set of events generated in the action of $t$. If $A$ is a sequential automaton, $s_A$ is a state of $A$, $b$ is a superstate of $s_A$, $s_b$ is a state of $b$, and $t_b$ is a transition of $b$, then:*

- $s_A \xrightarrow{RDD} s_b$ *iff* $D(s_b) \cap D(s_A) \neq \emptyset$, *or* $GE(s_A) \cap GE(s_b) \neq \emptyset$.
- $s_A \xrightarrow{RDD} t_b$ *iff* $D(t_b) \cap D(s_A) \neq \emptyset$, *or* $GE(s_A) \cap GE(t_b) \neq \emptyset$.

[Lan06] adopts the dependence relations defined by [WDQ02] but points out in [LH07] that parallel data dependence can produce a false dependency if the execution chronology is not taken into account. To get rid of these false dependencies, their tool applies the following rule after calculating parallel data dependencies: $x \overset{PDD}{\not\longrightarrow} y$ iff $x \xrightarrow{SO} y$, where $\xrightarrow{SO}$ is a Lamport-like [Lam78] happens-before relation on state machines. This happens-before relation is defined as using two other relations: the statechart concurrent relation and the statechart sequential order.

The authors in [Lan06] have also introduced a new data dependence relation called global data dependence in order for slicing a collection of statechart diagrams. By a collection of statechart models, they mean a statechart that may have many concurrent state machines at the top level. These are not dealt with by [WDQ02], who only deals with concurrency in sub-states. Global data dependence is defined between states and transitions with respect to global variables. Global variables are variables used by statecharts to communicate and they don't belong to any statechart but can be accessed by all.

**Definition 9 (Global Data Dependence for EHA)** *Let $A$, $B$ be two different statecharts. A state $s_B \in B$ or a transition $t_B \in B$ is global data dependent [Lan06] on a state $s_A \in A$ or transition $t_A \in A$ ( $s_A \xrightarrow{GDD} s_B$, $s_A \xrightarrow{GDD} t_B$, $t_A \xrightarrow{GDD} s_B$, $t_A \xrightarrow{GDD} t_B$) iff:*

- *some global output variables (defined in actions) of a state or transition of $A$ are used in the input of the state or transition of $B$;*
- *or some trigger events of a transition or state of $B$ are generated by a state or transition of $A$.*

The authors in [JJ06] have defined data dependence between variables found in boolean expressions of guards and atomic assignments of transitions of timed automata. Compared to Definition 2 and Definition 4, it is of finer granularity, i.e. it applies to parts of transitions rather than transitions leading to slices that can remove parts of transitions. Also, it applies to transitions of a set of timed automata that run in parallel.

**Definition 10 (Data Dependence for Timed Automata)** *Let the atomic assignments of actions $x$ and the boolean expressions of guards $y$ of a transition $t$ be called operations ($opers(t)$), thus $x \in opers(t)$ and $y \in opers(t)$. Let $t_1 \in T_i$, $t_2 \in T_j$, where $i, j = 1, .., n$ and $n$ refers to the number of timed automata that run in parallel. An operation $a_2$ in $t_2$ is data dependent [JJ06] on operation $a_1$ in $t_2$ if there is a variable $v$, which occurs in $a_1$ and $a_2$, if $v \in D(a_1) \cap U(a_2)$ as in Definition 2 and one of the following holds:*

1. *$t_1 = t_2$, $a_2$ follows $a_1$ (actions are sequences so follows is well defined [JJ06]) and $v \notin D(a_3)$ for any $a_3 \in opers(t_1)$ between $a_1$ and $a_2$;*
2. *there exists a path from $target(t_1)$ to the $source(t_2)$ such that $v$ is not re-defined in any operations in transitions contained in the path;*

Table 3: Papers that are categorised according to the type of control dependence definitions.

| **Event-Flow** | **Transition/Guard** | **Program-like** |
|---|---|---|
| Heimdahl and Whalen [HW97] | [WDQ02] | [KSTV03] |
| [CABN98] | [Lan06] | [JJ03] |
| | [SPH08] | [Oja07] |
| | | [LG08] |
| | | [ACH+09] |

*3.* $i \neq j$ *i.e. the automata are different.*

# 2 Control Dependence for Slicing FSMs

One of the challenges facing any attempt to slice a FSM is the problem of how to correctly account for control dependence. It is common for state machines modelling such things as non-terminating systems not to have a final computation point or 'exit state'. Moreover, FSM are interactive, i.e. there is an interplay between the environment and the model, where events are generated by the environment and trigger transitions in the model. Thus it is not only conditions that decide whether a transition occurs, but also whether a specific event occurs in the environment.

There have been three general approaches to defining control dependence. The first defines control dependence between events. We classify these definition as *Event-flow*. The second defines control dependence between transitions and guarding conditions and we classify these as *Transition/Guard*. The third defines control dependence similarly to how control dependence is defined by program slicing techniques. We call these definitions *Program-like*. The majority of the literature has focused on traditional definitions. In Table 3 the papers on FSM slicing are categorised according to the type of control dependence definition that they define.

## 2.1 Event-Flow Definitions

[HW97] were the first to define a control dependence-like definition for FSMs, in particular for RSML specifications. It differs from the traditional notion as it defines control flow in terms of dependency between events and generated events rather than as a structural property of the graph. Their definition can be applied to non-terminating systems or systems that have multiple exit nodes. However, it cannot be applied to any finite state machine, such as EFSMs, that do not generate events.

**Definition 11 (Control flow for RSML)** *Let $E$ be the set of all events and $T$ the set of all transitions. The relation $trigger(T \rightarrow E)$ represents the trigger*

*event of a transition. The relation* $action(T \rightarrow E^2)$ *represents the set of events that make up the action caused by executing a transition.* $follows(T \rightarrow T)$ *is defined as:* $(t_1, t_2) \in follows$ *iff* $trigger(t_1) \in action(t_2)$.

[CABN98] use the same control dependence relation as in Definition 11.

## 2.2 Transition/Guard Definitions

The authors in [WDQ02] have defined two control dependence definitions for EHA that are adopted by [LH07]: transition control dependence and refinement control dependence. Transition control dependence is defined between a state and transitions and can be applied to any FSM that may be non-terminating. They states that if a variable is defined in a state or transition $x$ and used in the guarding condition of a transition $t$ and not redefined on the path from $x$ to $t$, then $t$ is transition control dependent on $x$.

**Definition 12 (Transition Control Dependence (TCD))** *Let A be an EHA automaton, t a transition of A and* $CV(t)$ *be set of variables reference in the guard of a transition t.*

1. *If a variable is defined in a state v or a transition r and used in the guard of t and not redefined on the path from v or r to t, then* $v \xrightarrow{TCD} t$ *or* $r \xrightarrow{TCD} t$.
2. *If* $B, C$ *be two EHA automaton, s is a state in C and* $A, B$ *is a sub-state of s, q is a state in B, p is a transition in B, then* $t \xrightarrow{TCD} p$ *iff* $CV(t) \cap D(q) \neq \emptyset$ *(or* $CV(t) \cap D(p) \neq \emptyset$*).*

Refinement control dependence is defined between states of an automaton and its sub-sequential automaton. It can be applied to possibly non-terminating hierarchical FSMs.

**Definition 13 (Refinement Control Dependence)** *If state v is the initial state of a direct sub-sequential automaton of state u, then v is refinement control dependent on state u.*

These definitions of control dependence resemble data dependence, i.e. if the structure is flattened these correspond to data dependence as described in Definition 2. From the combined dependence relations defined by [WDQ02] (i.e. Definition 6, 7, 8, 12, 13 and 24), the hierarchical layer of an EHA for the sequential automaton of a state or transition can be determined. For example, if the sequential automaton of a state or transition $x$ is found on the $n^{th}$ layer, then $x$ depends on elements that belong to the sequential automata that are local in $(n-1)^{th}$ to $(n+1)^{th}$ layer of the EHA, which makes slicing efficient.

According to [SPH08], a variable assigned in a conditional assignment control depends on variables used in the conditions because these variables determine which branch of the condition is taken. This definition of control dependence also resembles data dependence.

Table 4: Classifying which FSM slicing approaches adopt or are based on NTICD and NTSCD.

| **Non-Termination Sensitive (NTSCD)** | **Non-Termination Insensitive (NTICD)** |
|---|---|
| [JJ03] | [KSTV03] |
| [Oja07] (NTSCD & DOD) | [ACH+09] |
| [LG08] | |

## 2.3 Program-like Definitions

In program slicing, two general types of control dependence definitions are described: *non-termination insensitive* and *non-termination sensitive*. Traditional control dependence, as used in [HRB90] is non-termination *insensitive*, with the consequence that the semantics of a program slice dominates the semantics of the program from which it is; slicing may remove non-termination, but it will never introduce it. A non-termination sensitive formulation was proposed as early as 1993 by [Kam93], but has not been taken up in subsequent slicing research. Non-termination sensitive slicing tends to produce very large slices, because all iterative constructs that cannot be statically determined to terminate must be retained in the slice, no matter whether they have any effect other than termination on the values computed at the slicing criterion. These 'loop shells' must be retained in order to respect the definition of non-termination sensitivity.

In moving slicing from the program level to the state based model level, the choice of whether FSM slicing should be non-termination sensitive or insensitive needs to be made. Both types of control dependence have been defined in the literature for FSM slicing and we consider each of these. The choice of which to use depends on the application of slicing. For example, non-termination sensitive control dependence (NTSCD) is desired when slicing FSMs for the purpose of model checking, as loops are kept and liveness properties can still be checked. Non-termination insensitive control dependence (NTICD) is preferred when slicing FSM for model comprehension as this will produce smaller slices. Table 4 classifies the papers on FSM slicing according to whether their control dependence definitions are non-termination sensitive or insensitive.

The following definitions of control dependence are given in terms of execution paths. Since a path is commonly presented as a (possibly infinite) sequence of nodes, a node is in a path if it is in the sequence. A transition is in a path if its source state is in the path and its target state is both in the path and immediately follows its source state. A *maximal path* is any path that terminates in an end node or final transition, or is infinite.

[KSTV03] present a definition of control dependence for EFSMs in terms of post dominance that requires execution paths to lead to an exit state. This definition captures the traditional notion of control dependence for static backward

slicing. However it can only determine control dependence for state machines with exactly one exit state, failing if there are multiple exit states or if the state machine is possibly non-terminating. For example, it can be applied to the ATM illustrated in Figure **??** but not to the ATM illustrated in Figure **??**.

**Definition 14 (Post Dominance [KSTV03])** *Let $Y$ and $Z$ be two states and $T$ be an outgoing transition from $Y$.*

- *State $Z$ post-dominates state $Y$ iff $Z$ is in every path from $Y$ to an exit state.*
- *State $Z$ post-dominates transition $T$ iff $Z$ is on every path from $Y$ to the exit state though $T$. This can be rephrased as $Z$ post-dominates $target(T)$.*

**Definition 15 (Insensitive Control Dependence (ICD))** *Transition $T_k$ is control dependent [KSTV03] on transition $T_i$ if:*

1. *$source(T_k)$ post-dominates transition $T_i$ (or $target(T_i)$), and*
2. *$source(T_k)$ does not post-dominate $source(T_i)$.*

In program slicing, [RAB+05] define control dependence for arbitrary CFGs (with a start node) of non-terminating programs, i.e. that may not have an exit node. They give definition for both *non-termination sensitive* (NTSCD) and *non-termination insensitive* control dependence (NTICD). The difference between these definitions lies in the choice of paths. NTSCD is given in terms of maximal paths, while NTICD is given in terms of control sinks (see Definition 21). This seminal work has inspired control dependence definitions for FSM models hereafter, except for the definition by [JJ03] . We first discuss all the NTSCD definitions and then NTICD definitions.

In [JJ03, JJ06] a NTSCD definition of control dependence is given for potentially non-terminating timed automata. Although they were the first to give such a definition for FSMs which is similar to the definition in [RAB+05] (both in terms of maximal paths), they have not been widely cited by other FSM slicing approaches.

In [JJ06], control dependence is defined in terms of post-dominance between states, where post-dominance is defined in terms of maximal paths and does not require a unique exit state like in Definition 14.

**Definition 16 (Post Dominance [JJ06])** *Let $X_1$ and $X_2$ be two states. State $X_1$ post-dominates state $X_2$ iff every maximal path from $X_1$ goes though $X_2$.*

**Definition 17 (NTSCD-JJ [JJ06])** *A state $S_2$ is control dependent on a state $S_1$ ($S_1 \xrightarrow{NTSCD} S_2$) in the same automaton, if $S_1$ does not post dominate (using Definition 16) $S_2$ and there is a path $\pi$ from $S_1$ to $S_2$ such that every state, except for $S_1$, in $\pi$ post dominates $S_2$.*

The earlier definition of control dependence given in [JJ03], differs from that given in [JJ06] as it includes an additional clause that states that all outgoing transitions from $S_1$ must have non-trivial guarding conditions. Thus the definition in [JJ03] is similar to Definition 20 but for states, rather than transitions.

[Oja07] adopts the NTSCD and decisive order dependence definitions, as in [RAB+05], which is given between nodes in a CFG. NTSCD cannot capture certain dependencies within loops, and hence Ranganath et al. defined decisive order dependence (DOD).

**Definition 18 (NTSCD [RAB+05] )** *In a CFG, a node $n_j$ is non-termination sensitive control dependent on a node $n_i$ ($n_i \xrightarrow{NTSCD} n_j$) iff $n_i$ has at least two successors $n_k$ and $n_l$ such that: for all maximal paths $\pi$ from $n_k$, where $n_j \in \pi$; and there exists a maximal path $\pi_0$ from $n_l$ where $n_j \notin \pi_0$.*

**Definition 19 (Decisive Order Dependence (DOD))** *Two nodes $p_1$ and $p_2$ are decisively order dependent [RAB+05] on $n$ ($n \xrightarrow{DOD} p_1, p_2$) if:*

1. *all maximal paths from $n$ contain both $p_1$ and $p_2$,*
2. *$n$ has a successor from which all maximal paths contain $p_1$ before $p_2$,*
3. *$n$ has a successor from which all maximal paths contain $p_2$ before $p_1$.*

[LG08][1] adapt Ranganath et al.'s NTSCD definition for Input/Output Symbolic Transition Systems (IOSTS). It is given in terms of transitions in an IOSTS model, rather than in terms of nodes in a CFG. The first clause of Definition 20 concerning the non-triviality of guards is introduced in order to avoid a transition being control dependent on transitions that are executed non-deterministically even though they are NTSCD control dependent. This clause prevents this property from being a purely structural property on graphs.

**Definition 20 (NTSCD-LG [LG08])** *A transition $T_j$ is control dependent on a transition $T_i$ if $T_i$ has a sibling transition $T_k$ such that:*

1. *$T_i$ has a non-trivial guard, i.e. a guard whose value is not constant under all variable valuations;*
2. *for all maximal paths $\pi$ from $T_i$, the source of $T_j$ belongs to $\pi$;*
3. *there exists a maximal path $\pi_0$ from $T_k$ such that the source of $T_j$ does not belong to $\pi_0$.*

FSM models differ from CFGs is several ways. They can have multiple start and exit nodes, more than two edges between two states and more than two successors from a state. Moreover, in CFGs, decisions (Boolean conditions) are made at the predicate nodes while in state machines they are made on

[1] Labbé et al.'s definition of control dependence in [LGP07] differs slightly from Labbé et al. [LG08], so we evaluate the most recent.

transitions. Labbé et al. take such differences into account when adapting NTSCD.

[ACH+09] have defined a new control dependence definition by extending Ranganath et al.'s NTICD definition and subsuming Korel et al.'s definition in order to capture a notion of control dependence for EFSMs that has the following properties. First, the definition is general in that it should be applicable to any reasonable FSM variant. Second, it is applicable to non-terminating FSMs and / or those that have multiple exit states. Third, by choosing FSM slicing to be non-termination insensitive (in order to coincide with traditional program slicing) it produces smaller slices than traditional non-termination sensitive slicing.

Following [RAB+05], the paths considered are sink-bounded paths, i.e. those that terminate in a control sink as in Definition 21. A control sink is a region of the graph which, once entered, is never left. These regions are always SCCs, even if only the trivial SCC, i.e. a single node with no successors.

**Definition 21 (Control Sink)** *A control sink, $\mathcal{K}$, is a set of nodes that form a strongly connected component such that, for each node $n$ in $\mathcal{K}$ each successor of $n$ is in $\mathcal{K}$.*

Unlike NTICD, sink-bounded paths are unfair, i.e. we drop the fairness condition in the Ranganath et al.'s definition of sink paths. For non-terminating systems this means that control dependence can be computed within control sinks.

**Definition 22 (Unfair Sink-bounded Paths)** *A maximal path $\pi$ is sink-bounded iff there exists a control sink $\mathcal{K}$ such that $\pi$ contains a transition from $\mathcal{K}$.*

**Definition 23 (Unfair Non-termination Insensitive Control Dependence)** *(UNTICD) A transition $T_j$ is control dependent on a transition $T_i$ iff:*

1. *for all paths $\pi \in UnfairSinkPaths(target(T_i))$, the $source(T_j)$ belongs to $\pi$;*
2. *there exists a path $\pi \in UnfairSinkPaths(source(T_i))$ such that the $source(T_j)$ does not belong to $\pi$.*

UNTICD is in essence a version of NTICD modified to EFSMs (rather than CFGs) and given in terms of unfair sink-bounded paths.

Table 5 compares the transitive closure of control dependence definitions given in terms of transitions. Note that $ICD^*$ denotes the transitive closure of $ICD$. Similarly for $NTSCD-JJ^*$, $NTSCD-LG^*$, and $UNTICD^*$. NTSCD and DOD as given by [Oja07] are not in the table as they are defined between states, rather than transitions.

Table 5: Comparison of transitive closure of control dependence definitions.

| **Definition** | **Comparison of Transitive Closures** |
|---|---|
| ICD [KSTV03] | ICD$^*$ $\subseteq$ NTSCD-JJ$^*$ |
| NTSCD-JJ [JJ06] | UNTICD$^*$ $\subseteq$ NTSCD-JJ$^*$ |
| NTSCD-LG [LG08] | NTSCD-LG$^*$ $\subseteq$ NTSCD-JJ$^*$ |
| UNTICD [ACH$^+$09] | UNTICD$^*$ $\subseteq$ NTICD-JJ$^*$ |

# 3 Interference Dependence for Slicing FSMs

Most FSM slicing approaches handle communication and synchronisation by introducing new dependencies, such as interference dependence. As with slicing concurrent programs, the computation of interference dependencies can be complex, if the possible orders of execution must be considered to compute *precise* dependencies [Kri03]. Even if the computed dependencies are precise, the slicing algorithm can be imprecise if it just assumes transitivity of the dependencies and traverses the reachable dependencies. As we discuss below, only a few FSM slicing approaches try to compute precise dependencies.

The authors in [WDQ02] have defined synchronisation dependence between transitions and states. It states that if the trigger event of some transition in an element (state or transition) $x$ is generated by the action of an element $y$ and the automaton which $x$ and $y$ belong to are concurrent, then $x$ is synchronisation dependent on $y$. However, their slicing algorithm traverses the dependencies (including the various data and control dependencies) and assumes transitivity and thus is imprecise.

**Definition 24 (Synchronisation Dependence (SD))** *A state $s_A$ or transition $t_A$ is synchronisation dependent on a concurrent state $s_B$ or transition $t_B$ ($s_A \xrightarrow{SD} s_B$, or $s_A \xrightarrow{SD} t_B$, or $t_A \xrightarrow{SD} s_B$, or $t_A \xrightarrow{SD} t_B$) iff some events generated by the latter are used as trigger events of the other.*

The authors in [LH07] have adopted Definition 24 but also introduce a new dependence relation for a collection of statecharts. Global synchronisation dependence is similar to Definition 24 except that it is between statecharts and involves global generated events.

**Definition 25 (Global Synchronisation Dependence (GSD))** *Let $A$ and $B$ be two different statecharts. A state $s_A$ or transition $t_A$ is global synchronisation dependent on a state $s_B \in B$ or transition $t_B \in B$ ($s_A \xrightarrow{GSD} s_B$, or $s_A \xrightarrow{GSD} t_B$, or $t_A \xrightarrow{GSD} s_B$, or $t_A \xrightarrow{GSD} t_B$) iff some global events generated by the latter are used as the trigger of the other.*

They use a Lamport-like [Lam78] happens-before relation to ensure that the dependencies exist only between states and transitions where the source state

or transition can happen before the target state or transition. Although this increases the precision of the dependence relations, the slicing algorithm is based on the (transitive) traversal of the dependencies and thus is imprecise.

The authors in [GR02] have also introduced dependencies to model and slice Argos specifications: *transition*, *hierarchy* and *trigger* edges model the dependencies between states and sub-states. Their slicing algorithm traverses the dependencies with no special handling of intransitivity.

The authors in [Oja07] have defined interference dependence for UML state machines between parts of transitions. This definition is similar to the one in [HCD+99] for multi-threaded Java programs. Their slicing algorithm is based on traversal of the dependencies and thus is imprecise.

**Definition 26 (Interference Dependence [Oja07])** *A node $n_j$ is interference dependent on a node or parameter $n_i$ if $v \in U(n_j)$, $w \in D(n_i)$, $v = w$ and the access to $v$ and $w$ are not local to $n_j$ or $n_i$ ($n_i$ and $n_j$ are in different state machines or different instances of the same state machine).*

The authors in [LG08] have defined a communication dependence relation for communicating automata (IOSTSs) that identifies dependencies owing to communicating actions. These dependencies are inter-automata, unlike their data and control dependencies (Definition 4 and Definition 20) that are intra-automata. A communication dependence is defined between two transitions $t_1$ and $t_2$ in two different IOSTSs if there exists a channel that potentially allows a data or control flow to occur between $t_1$ and $t_2$.

**Definition 27 (Communication Dependence [LG08])** *Transitions $t_i$ and $t_j$ are communication dependent iff there exists a channel $c$ such that:*

1. *The action $a_j = c?x$ occurs and the system waits on channel $c$ for the reception of a value to be assigned to the attribute variable $x$; and b) the action $a_i = c!t$ occurs for the system to emit a message, with $t$ as argument, on the channel $c$.*

2. *The action $a_j = c?$ occurs and the system waits for a signal to occur on the channel $c$; and b) The action $a_i = c!$ occurs and the system emits a message on channel $c$ with no arguments.*

The authors in [LG08] have presented a slicing algorithm based on traversing the dependencies, however, as communication dependence is not transitive, they accept the reduced precision. [GR08] have compiled statecharts into Java programs which are then sliced dynamically. These types of approaches are different to those discussed in this section as they analyse concrete executions and reduce the machine according to a specific test case similar to dynamic program slicing. Because the synchronization and communication can directly be observed and don't have to be approximated by static analysis, concurrency and communication don't cause problems there.

# 4 Other Dependence Relations for Slicing FSMs

The authors in [JJ06] have defined two new dependencies: *clock* and *time* dependence. Their slicing algorithm is based on traversal of the dependencies and thus is imprecise. Clock dependence is defined between two states in the same timed automaton.

**Definition 28 (Clock Dependence)** *A state $s_2$ is clock dependent on a state $s_1$ in the same timed automaton if one of the following holds:*

1. *there exists a clock $x$ in the set of clocks of the automaton and a transition $t$, where $s_1 = source(t)$, such that $x$ is defined in the clock assignment of $t$ and is in the set of clocks of $s_2$, and $target(t) = s_2$, or*

2. *there exists a path $\pi$ from $target(t)$ to $s_2$ such that the clock assignment for all transitions $t'$ contained in $\pi$ with respect to $x$ is equal to $x$.*

Time dependence is defined between two states, $s_1$ and $s_2$, of the same process if $s_2$ is reachable from $s_1$ and time has elapsed in $s_2$.

**Definition 29 (Time Dependence)** *For two states $s_1, s_2$ in the same automaton, $s_2$ is time dependent on $s_1$ if $s_2$ is reachable from $s_1$ and:*

1. *all transitions going out of $s_2$ (i.e. $source(t) = s_2$) are urgent and always one transition is enabled. Heuristics are used to check the last condition;*

2. *the state invariant has a constraint of the form $x = 0$, where $x$ is a clock and the clock assignments of all incoming transitions set $x$ to $x_0$.*

*If the given conditions are violated then time in state $s_2$ cannot lapse.*

The authors in [SPH08] have defined adaptive dependence, whereby a functional variable is influenced by the adaptive variables occurring in the configuration guards. The adaptive variables in the configuration guards determine whether the functional assignments are executed.

# References


[ACH+09] Kelly Androutsopoulos†, David Clark, Mark Harman, Zheng Li, and Laurence Tratt. Control dependence for extended finite state machines. In *Fundamental Approaches to Software Engineering (FASE), Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS*, York, UK, March 2009. Springer Berlin/Heidelberg.

[CABN98] William Chan, Richard J. Anderson, Paul Beame, and David Notkin. Improving efficiency of symbolic model checking for state-based system requirements. *SIGSOFT Software Engineering Notes*, 23(2):102–112, 1998.

[FL05] Chris Fox and Arthorn Luangsodsai. And-or dependence graphs for slicing statecharts. In *Beyond Program Slicing*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[GR02] Vinod Ganapathy and S. Ramesh. Slicing synchronous reactive programs. In *Electronic Notes in Theoretical Computer Science, 65(5). 1st Workshop on Synchronous Languages, Applications, and Programming*, Grenoble, France, April 2002. Elsevier.

[GR08] Liang Guo and Abhik Roychoudhury. Debugging statecharts via model-code traceability. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008*, pages 292–306, Port Sani, Greece, 2008. Springer Berlin/Heidelberg.

[HCD+99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Symposium on Static Analysis*, pages 1–18, London, UK, 1999. Springer-Verlag.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[HTW98] Mats P. E. Heimdahl, Jeffrey M. Thompson, and Michael W. Whalen. On the effectiveness of slicing hierarchical state machines: A case study. In *EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO*, pages 10435–10444, Washington, DC, USA, 1998. IEEE Computer Society.

[HW97] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *Proc. Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, 1997. Springer–Verlag.

[JJ03] Agata Janowska and PawełJanowski. Slicing timed systems. *Fundam. Inf.*, 60(1-4):187–210, 2003.

[JJ06] Agata Janowska and PawełJanowski. Slicing of timed automata with discrete data. *Fundamenta Informaticae, SPECIAL ISSUE ON CONCURRENCY SPECIFICATION AND PROGRAMMING*, 72(1-3):181–195, 2006.

[Kam93] Mariam Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, July 2003.

[KSTV03] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state based models. In *IEEE International Conference on Software Maintenance (ICSM'03)*, pages 34–43, Los Alamitos, California, USA, September 2003. IEEE Computer Society Press.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lan06] Sara Van Langenhove. *Towards the Correctness of Software Behavior in UML A Model Checking Approach based on Slicing*. PhD thesis, Ghent University, May 2006.

[LG08] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20(6):563–595, 2008.

[LGP07] Sebastien Labbe, Jean-Pierre Gallois, and Marc Pouzet. Slicing communicating automata specifications for efficient model reduction. In *Proceedings of ASWEC*, pages 191–200, USA, 2007. IEEE Computer Society.

[LH07] Sara Van Langenhove and Albert Hoogewijs. $SV_tL$: System verification through logic tool support for verifying sliced hierarchical statecharts. In *Lecture Notes in Computer Science, Recent Trends in Algebraic Development Techniques*, pages 142–155, Berlin / Heidelberg, 2007. Springer.

[Oja07] Vesa Ojala. A slicer for UML state machines. Technical Report HUT-TCS-25, Helsinki University of Technology Laboratory for Theoretical Computer Science, Espoo, Finland, 2007.

[RAB+05] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP*, pages 77–93, Berlin, 2005. Springer-Verlag.

[SPH08] Ina Schaefer and Arnd Poetzsch-Heffter. Slicing for model reduction in adaptive embedded systems development. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 25–32, NY, USA, 2008. ACM.

[WDQ02] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking UML statecharts. In *Proceedings of*

*the 4th International Conference on Formal Engineering Methods (ICFEM)*, pages 435–446, UK, 2002. Springer-Verlag.