2013 IEEE 37th Annual Computer Software and Applications Conference Workshops

Using Dependency Relations to Improve Test Case Generation from UML Statecharts

Valentin Chimisliu
Institute for Software Technology
University of Technology Graz
Graz, Austria
chimisliu@ist.tugraz.at

Franz Wotawa
Institute for Software Technology
University of Technology Graz
Graz, Austria
wotawa@ist.tugraz.at

***Abstract*—In model-based testing the size of the used model has a great impact on the time for computing test cases. In model checking, dependence relations have been used in slicing of specifications in order to obtain reduced models pertinent to criteria of interest. In specifications described using state based formalisms slicing involves the removal of transitions and merging of states thus obtaining a structural modified specification. Using such a specification for model based test case generation where sequences of transitions represent test cases might provide traces that are not valid on a correctly behaving implementation. In order to avoid such trouble, we suggest the use of control, data and communication dependences for identifying parts of the model that can be excluded so that the remaining specification can be safely employed for test case generation. This information is included in test purposes which are then used in the test case generation process. We present also first empirical results obtained by using several models from industry and literature.**

***Keywords*-Test Case Generation, UML Statecharts, Control Dependence, Data dependence, Communication Dependence**

I. Introduction

Model-based test-case generation assumes the availability of a model describing the desired behavior of the system under test (SUT). In our setting the model is constructed by means of the modeling language UML [1]. The model itself describes a distributed system that uses asynchronous communication. Because the UML lacks a formal semantic (it does have a formal syntax however), we automatically extract [2] a formal description of the desired behavior of the SUT in the form of a LOTOS [3] specification from the available UML model. LOTOS is a formal description technique developed within ISO for the formal specification of open distributed systems. The advantage of this transformation is that we are able to make use of already existing tools in order to automatically perform verification and validation tasks. In the current work, we are interested in the generation of conformance test cases by using the TGV [4] test-case generator from the CADP toolbox.

In order to synthesize test cases, TGV requires a formal specification of the SUT (the LOTOS specification in our case). It also requires test purposes in order to focus the generation of test cases on particular aspects of the system. A test purpose represents an abstraction of the original model describing a scenario of interest, which should be tested. Test purposes are represented as Input Output Labeled Transition Systems (IOLTS) and make use of special predefined labels in order to control the test case generation process. One of these is the "$REFUSE$" label, which is used to mark parts of the model that should not be explored during a particular test-case generation process.

Because TGV makes use of enumerative techniques (an IOLTS representing the semantic of the LOTOS specification is generated on the fly) for test-case generation it is obvious that the presence of more refuse transitions leads to a faster computation of test cases. Hence, the chances of running into the state space explosion problem are reduced.

In this paper, we aim at using different dependence relations in order to automatically identify parts of the models that have no influence on targeted transitions and thus can be omitted during the generation process. This is done by computing direct and indirect dependences for the transitions we aim to cover with the generated test cases. The dependence information is used to insert refuse transitions in the test purposes, and thus reducing the searched state space during the generation process.

The rest of this paper is organized as follows. In Section II we describe the UML modeling assumptions and a running example, which is followed by Section III where we define the used dependences and introduce the algorithms for computing them. In Section IV we describe the test purpose generation process, and in Section V we discuss first empirical results obtained using several case studies. Finally, we discuss related work in Section VI and conclude this paper in Section VII.

II. System Under Test

The UML statechart [1] diagrams belong to the UML behavioral diagrams. They are used to describe dynamic aspects by defining different states of a system. The change from one state to another is usually controlled by external or internal events.

The behavior of the considered type of systems is described by means of asynchronously communicating UML statecharts.

Our running example depicted in Figure 1, which describes the diagnosis functionality of modern vehicles. Its

978-0-7695-4987-3/13 $26.00 © 2013 IEEE
DOI 10.1109/COMPSACW.2013.24

purpose is to store the type, occurrence, and origin of errors during operation of the vehicle.

Besides the diagnosis functionality our system also contains a model describing the behavior of the ignition switch of the vehicle and two other models defining the conditions needed for errors to be detected.

The diagnosis statechart consists of five states and accepts four messages namely *evErrorActive*, *evErrorNotActive*, *evRequestErrorMemoryClear* and *evSetGwUnlearnCounterCopy*.

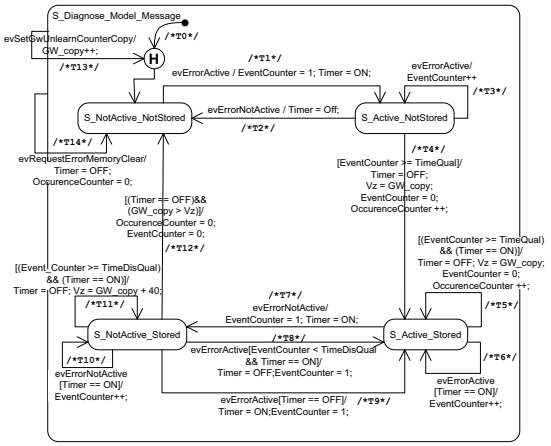

Figure 1. Statechart of Diagnosis Functionality

The state *S_NotActive_NotStored* corresponds to normal functioning when no error is detected. After an error is detected, the system moves to the state *S_Active_NotStored*, which means that an error has been detected but is not yet stored. The error is stored after receiving five *evErrorActive* events, and the system moves to the *S_ActiveStored* state. This means that the error has been stored. The diagnosis module shall leave this state and move to the *S_NotActive_Stored* state only after receiving an *evErrorNotActive* event.

As already mentioned in Section I we use this model in order to automatically derive a LOTOS specification. Since the transformation has already been presented in [2], here we only mention the first step of the transformation, which consists of the flattening of the statecharts. This is important for the current approach since the computation of the dependences is carried out on the flattened representation of the statechart.

The flattening process removes the hierarchical structures and the pseudostates.Figure 2 presents the flattened representation of the diagnosis statechart from Figure 1. Due to readability reasons the labels of the transitions in Figure 2 have been omitted, however they remain the same as in the original model. During the flattening process transition copies are created for the transitions originating from composite states. Each such transition generates a copy of itself for each state contained by the composite state.

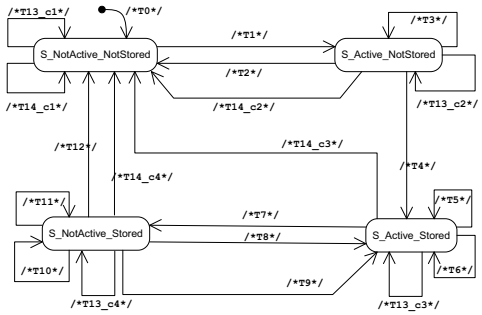

Figure 2. Flattened Representation of Diagnosis

In our running example (Figures 1 and 2) transition $T14$ generates the copies $T14\_c1$, $T14\_c2$, $T14\_c3$ and $T14\_c4$. A detailed description of the flattening process can be found in [5].

III. Dependences

A. Control Dependence

Informally, in classical definitions of control dependence of sequential programs a statement s_j is control dependent on a statement s_i if statement s_i causes the execution of statement s_j.

State based formalisms differ from sequential programs so control definitions have also been adapted for such formalisms. In [6] two such definitions are given. The one closest to our setting is called Non-termination Sensitive Control Dependence (NTSCD) (Definition 2) and is given in terms of maximal paths (Definition 1).

Definition 1: **(Maximal Path).** A path π is maximal if it terminates in an end state (state with no outgoing transitions) or is infinite.

1) Computing Control dependence: The usage of maximal paths (especially infinite ones) is supported by the observation that a potentially infinite execution of a loop might impede the execution of other transitions. Another important observation of [6] (also acknowledged in [7]) is that reaching a start node in a reactive system is analogous to reaching an end node in a program, i.e., the behavior will start over again. Thus in such cases the start node can also be used as an end state when computing maximal paths.

Definition 2: **(Non-termination Sensitive Control Dependence (NTSCD)).** $t_i \xrightarrow{NTSCD} t_j$ means that t_j is non termination sensitive control dependent on a transition t_i iff t_i has at least one sibling t_k such that:

1) for all paths $\pi \in MaximalPahts(target(t_i))$, the $source(t_j)$ belongs to π;
2) there exists $\pi \in MaximalPahts(source(t_k))$ a path such that $source(t_j)$ does not belong to π.

Informally a transition t_j is control dependent on transition t_i if the execution of t_i will always lead to $source(t_j)$ and the execution of a sibling transition t_k of t_i (a maximal path) might not lead to $source(t_j)$ (t_j might not be executed). Considering the flattened representation of the

Input: $SC = (S_s, T, V, st_i)$
Output: CD(T), PI(T)
1: **for all** $t \in \{t | t \in T(SC) \wedge \exists t_s \in sibling(t) : target(t_s) \neq target(t)\}$ **do**
2: **for all** $s_c \in \{s_c | \exists t_i \in \bigcap maxPaths(t) : source(t_i) = s_c \vee target(t_i) = s_c\}$ **do** // for common nodes on all paths
3: **if** $\forall t_{st} \in sibling(t) \exists \pi \in maxPaths(t_{st}) : \forall t_i \in outgTr(s_c) \rightarrow t_i \notin \pi$ **then**
4: **for all** $t_o \in \{t_o | t_o \in T(SC) \wedge source(t_o) = s_c\}$ **do**
5: $CD(t_o) \leftarrow CD(t_o) \cup t$
6: **for all** $t_{st} \in \{t_{st} | t_{st} \in sibling(t) \wedge \exists \pi \in maxPaths(t_{st}) : \forall t_i \in outgTr(s_c) : t_i \notin \pi\}$ **do**
7: $PI(t_o) \leftarrow PI(t_o) \cup t_{st}$
8: **end for**
9: **end for**// added NTSCD controlling and potential independent transitions
10: **end if**
11: **end for**
12: **end for**

Figure 3. NTSCD Computation Algorithm

diagnosis model (Figure 1) the set of NTSCD relations contains the dependence: $T1 \xrightarrow{NTSCD} T2$. Thus $T1$ will always cause the source state of $T2$ - *S_Active_NotStored* to be entered and there exists the maximal path formed by the loop $T13\_c2$ that does not contain *S_Active_NotStored*.

For computing the NTSCD relations we apply the algorithm in Figure 3 for every flattened statechart $SC = (S_s, T, V, st_i)$ in our model. Where S_s is the set of simple states , T the set of transitions, V the set of variables used in the statechart and st_i is the initial state of the statechart.

Because we consider maximal paths (which include also infinite paths) we need to identify cycles in our model SC. Thus this information is contained in the set $CYCLES(SC)$ whose elements are sets of states representing the cycles in the model SC.

For transitions that are not part of a cycle, all nodes of the cycle that are targeted by transitions whose source does not belong to the cycle can be considered end states. Thus we define $SINKS(t)$ (equation 1) as the set containing all such states, end states and also the initial state st_i (according to the observation in Section III-A). In the algorithm we also make use of the function $maxPaths(t)$ (equation 2) which provides all maximal paths starting with transition t.

$$SINKS(t) = \{s | s \in S_s(SC) \wedge |outTr(s)| = 0\} \cup \{st_i\} \cup \{s | s \in S_s(SC) \wedge \exists C \in CYCLES(SC) : (s \in C \wedge source(t) \notin C)\} \quad (1)$$

$$maxPaths(t) = \{(t_1, t_2, ...t_n) | t_1 = t \wedge target(t_n) \in SINKS(t)\} \quad (2)$$

The algorithm for computing the NTSCD (Figure 3) requires as input a flattened statechart $SC = (S_s, T, V, st_i)$ and provides as output two key value maps:

1) CD(T) - key value map containing as key a transition and as value a set of transitions on which t is control dependent on;
2) PI(T) - key value map containing as key a transition and as value transitions that potentially do not influence t (from the NTSCD point of view).

The transitions with at least one sibling (statement 1) might NTSCD control other transitions. Only transitions originating from the states that appear in all maximal paths (statement 2) of transition t might be control dependent on t.

If there exists at least a sibling t_{st} of transition t that has at least one maximal path, which does not contain the considered state s_c, all the outgoing transitions t_o of s_c ($outgTr(s_c)$) are NTSCD control dependent on transition t (statements 4 - 5). Since we need as test cases sequences of transitions that are valid on the specification, the transitions on maximal paths starting at t and containing s_c are also added to the list of transitions $outgTr(s_c)$. For simplicity reasons this is not explicitly depicted in the algorithm. Also all the sibling of t that possess at least one maximal path bypassing s_c are added as potential independent transitions for t_o (statements 6 - 7).

B. Data Dependence

Classical data dependence definitions are given in terms of variable definitions and uses. Thus in terms of EFSM a variable is used on a transition if its value appears in the guard of the transition or appears on the right side of an assignment in the action of the transition. A variable is defined if it is assigned a value when the respective transition is fired.

We adopt the data dependence definition of [8] since the used formalism of EFSM is very similar to the representation we obtain after the flattening of the statecharts.

Definition 3: **(Data Dependence(DD)).** $t_i \xrightarrow{DD}_v t_k$ means that transition t_i and t_k are data dependent with respect to variable v if:

1) $v \in D(t_i)$, where $D(t_i)$ is a set of variables defined by actions of transition t_i;
2) and $v \in U(t_k)$, where $U(t_k)$ is a set of variables used in the guard and actions of transition t_k;

3) and there exists a path in the EFSM from (t_i) to the $target(t_k)$ whereby v is not modified.

Since we do not modify the structure of the specification, besides the data dependence between a transition t_i and t_k with respect to a variable v we are also interested in the definition free paths, i.e., paths from t_i to t_k along which v is not redefined. We compute these paths by using depth first search to explore the model backwards starting from t_k and following the incoming transitions of $source(t_k)$. Each time a variable v used by t_k is defined we save the respective path and add its transitions to the set of transitions t_k is dependent on. The considered paths are all simple paths. Thus after the execution of the algorithm the map $DD(t_k)$ will contain the transitions that t_k depends on.

Since we are only interested in the execution of certain transitions, we reduce the set of variables of interest to the ones used in the guards of the transitions (including the variables that directly or indirectly influence them). The rationale behind this is the fact that the truth value of the guards is the one that allows for the execution of the transitions. Thus we reduce the set $DD(t_k)$ to the set of transitions that directly or indirectly might influence the truth value of the guard of t_k.

C. Communication Dependence

Depending on the state based formalism used, there are several definitions for communication dependence under different names. Out of these, the one closest to what we need in our setting is the one given by [9] also called synchronization dependence. This definition is more general and is given in terms of states and transitions in concurrent models. Informally it states that if the trigger event of some transition in an element x (x can be a state or transition) is generated by the action of an element y, and the automata of x and y are concurrent, then x is synchronization-dependent on y.

In our particular case the communication dependence only relates to transitions within concurrent models. Thus we adapt the definition of [9] to Definition 4.

Definition 4: **(Communication Dependence(COMD))**. Given two transitions $t_i \in T(SC_1)$ and $t_k \in T(SC_2)$, $t_i \xrightarrow{COMD} t_k$ means that transition t_k is communication dependent on t_i iff:

1) SC_1 and SC_2 are two concurrent statecharts;
2) $trigger(t_k)$ - the triggering event of t_k is generated by the actions of t_i.

The direct communication dependences are computed by iterating through the transitions whose actions generate events and adding these transitions to the key value map $COMD(t)$ where t is the transition triggered by the generated event.

D. Computing Independent Transitions

After computing the direct dependences for each model in our specification we compute the indirect dependences given a set of transitions of interest. Informally transition

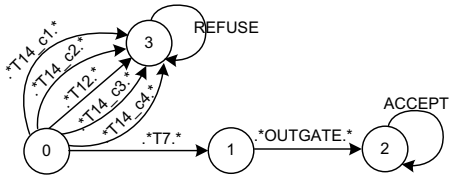

Figure 4. Test Purpose for T7

t_j is indirectly dependent on transition t_i if there exists a sequence of dependences leading from t_i to t_j (Definition 5 adapted from [7]). This represents the transitive closure between t_i and t_j considering the ID relation.

Definition 5: **(Indirect Dependence (ID)).** $t_i \xrightarrow{ID} t_j$ means that t_j is indirectly dependent on t_i iff there exists a sequence $(t_1, ..., t_k)$ where $t_1 = t_i$ and $t_k = t_j$ such that for all $1 \leq n \leq k$: $t_n \xrightarrow{NTSCD} t_{n+1}$ or $t_n \xrightarrow{DD} t_{n+1}$ or $t_n \xrightarrow{COMD} t_{n+1}$.

After computing the transitive closure for a transition t we obtain the set T_{CTRL} containing all transitions t_i such that $t_i \xrightarrow{ID} t$. Thus $T_{IND}(t) = PI(t) \setminus T_{CTRL}(t)$ is the set containing the transitions that do not influence (directly or indirectly) transition t. $PI(t)$ contains the transitions potentially not controlling t (computed with the algorithm in Figure 3).

IV. Test Purpose Generation

Due to the fact that during the transformation we preserve the traceability between the UML model and the LOTOS specification we are able to generate test purposes aimed at covering the original UML model. Thus, Figure 4 contains the IOLTS representation of a test purpose generated for covering transition *T7* of the Diagnosis model. Figure 5 shows the test case generated for the above mentioned test purpose.

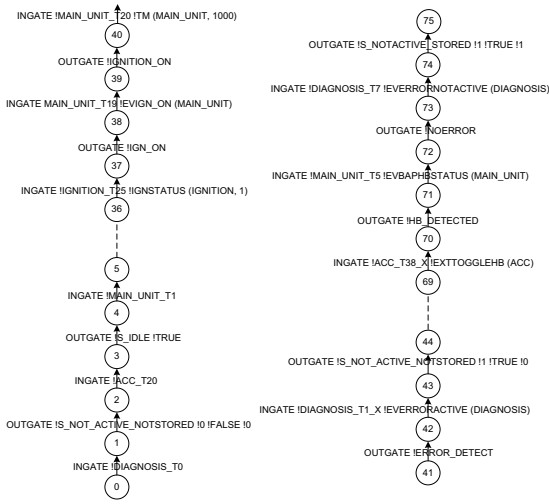

Figure 5. Test Case Covering Transition T7

Table I
COVERAGE AIMED TEST CASE GENERATION RESULTS

$Model$	$Approach$	TPs	$ValidTPs$	TCs	$DepCmpt$	$CompTime$	$TCov_{flat}$	$TCov$
Flasher	Dep.	72	70	70	8s	14m30s	97%	100%
	Apr1	72	70	70	-	58m10s	97%	100%
Diagnosis	Dep.	44	42	39	2s	1h20m	95%	97%
	Apr1	44	42	38	-	1h43m50s	95%	97%
KeylessEntry	Dep.	43	39	39	2s	2m42s	91%	100%
	Apr1	43	39	39	-	2m38s	91%	100%
MicrowaveOven	Dep.	37	37	37	1s	2m10s	100%	100%
	Apr1	37	37	36	-	27m20s	97%	97%
LoanApprovalWS	Dep.	22	22	22	1s	1m9s	100%	100%
	Apr1	22	22	22	-	1m20s	100%	100%

The refuse transitions in Figure 4 are identified by computing the independent transitions for each transition in the model.

In the test purpose definition, labels of transitions are denoted by strings that can also be specified by using regular expressions (e.g. *".*"* or *".*T7.*"*). The label *".*T7.*"* contains the ID of the searched transition whereas *".*OUTGATE.*"* represents the action needed to get the values of the variables used in the statechart after triggering the transition *T7*.

The edges leading to state 3 are labeled with the IDs of the transitions *T7* does not depend on. The edge with the label "$REFUSE$" is used to mark parts of the model that will not be explored during the test generation process. Basically edges in the specification whose labels fit (the regular expression of) labels on edges in the test purpose leading to the source of the $REFUSE$ edge will not be explored (and thus neither the behavior that they lead to). The generation process will stop as soon as an action matching the expression *".*T7.*"* followed by one matching *".*OUTGATE.*"* is encountered.

V. EXPERIMENTAL RESULTS

We evaluated the proposed approach using three real-world examples (Flasher, Diagnosis and KeylessEntry) originating from the automotive domain and four more from literature.

Transition coverage on the flattened model has the advantage that it subsumes transition coverage on the original model. This comes from the fact that it tries to cover all copies of a transition T generated during the flattening process. On the original model this is equivalent to firing T from every state contained by its source state.

However, a drawback in trying to cover the flattened model is the fact that some transition copies are not reachable in the flattened model. This originates from the fact that certain combinations between simple states and transition copies are not possible.

In Table I we present the results obtained when using the current test purpose generation technique aimed at transition coverage.

The first column of the table contains the name of the model for which the test purposes were generated. Column $Approach$ contains the test case generation approach where *Dep.* stands for the current approach and *Apr1* for the previous one [10]. The next column TPs contains the total number of generated test purposes. Column $ValidTPs$ presents the number of valid test purposes. By valid test purpose we mean test purposes not targeting transitions copies that are not reachable on the flattened model.

In column TCs we give the number of generated test cases. We imposed a limit of 25 minutes per test purpose. If no test case was generated within this time, the generation process is stopped. Column $DepCmpt$ contains the time needed for computing the dependence relations while column $CompTime$ contains the time TGV was allowed to run for the generation process.

The last two columns ($TCov_{flat}$ and $TCov$) contain the transition coverage on the flattened and non flattened model respectively.

For most of the models, the current approach delivered better results than the previous one. This was to be expected because one eliminated transition might translate to a (more or less) large part of the behavior (at IOLTS level - the enumerated behavior of the specification) that is not considered during the test case generation process.

In some cases (*Diagnosis* and *MicrowaveOven*) the dependences helped in finding transitions that the old approach was not able to find.

Even equipped with refused transitions the current approach failed in finding test cases to cover three transitions in the *Diagnosis* model. However the previous approach failed in finding four such test cases. Also in this case the current approach outperforms the old one.

VI. RELATED WORK

Slicing has been used in [11] for the purpose of test case generation from UML activity diagrams. They generate test cases aimed at path coverage by computing dynamic slices corresponding to each conditional predicate on the edges of the diagram. In their work no static control dependences are used and only data dependences are employed so that only the nodes that affect the truth value of the predicate on the edge at run time are kept in the slices.

The approach from [12] is the closest to the one we present in this article. There test purposes are generated

and extended with refuse states. In that work the refused states are computed using data flow graphs that are extracted from LOTOS specifications. The dependence relations (data flow graphs), type and behavioral semantic (synchronously communicating processes) of the systems are some of the differences to the current approach.

An approach using slicing for test case generation is [13], where the authors compute slices from specifications given in the formal language IF. The slices are calculated with respect to sets of signals (inputs or outputs) and also require external data in the form of test purposes or feeds. Our approach uses different formalisms and there is no need for external user provided data. We also do not generate a new specification for each criterion.

In [14] an approach to derive test purposes from temporal logic properties specifications is proposed. The approach uses modified model checking algorithms to extract examples and counterexamples from the state space of the specification. Test purposes are then constructed by analyzing the extracted behaviors.

VII. Conclusions and Future Work

In this article we present the usage of different dependence relations in order to enhance test purposes with refuse states. These states are used by the TGV test-case generation tool in order to limit the searched state space during the generation process. We use these refuse states in order to improve a previously presented test case generation technique [10] aimed at structural coverage (state and transition coverage) of a specification given in terms of asynchronously communicating statecharts.

We evaluated the proposed approach using a case study comprising three real world examples from the automotive domain. The obtained results show an improvement from the initial version of the test case generation technique. However, the approach still needs further evaluation by using it on a larger class of specifications and identifying properties indicating its usefulness.

Another direction of interest for future work is the investigation of the happens-before relation in case of communication dependence.The happens-before relation [15] helps by ensuring that dependences exist only between transitions where the source transition can happen before the target transition. In our case this means a possible increase of the number of identified refuse transitions.

Acknowledgement

The research herein is partially conducted within the competence network Softnet II Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

References

[1] "Unified modeling language UML 2.0," Object Management Group OMG. [Online]. Available: http://www.omg.org/spec/UML/2.0/

[2] V. Chimisliu and F. Wotawa, "Abstracting timing information in UML statecharts via temporal ordering and LOTOS," in *Proc. of the 6th International Workshop on Automation of Software Test*, ser. AST '11. ACM, 2011, pp. 8–14.

[3] ISO, "ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour," 1989.

[4] C. Jard and T. Jéron, "TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 297–315, 2005.

[5] C. Schwarzl and B. Peischl, "Test sequence generation from communicating UML state charts: An industrial application of symbolic transition systems," *Quality Software, International Conference on*, pp. 122–131, 2010.

[6] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer, "A new foundation for control dependence and slicing for modern program structures," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 5, 2007.

[7] S. Labbé and J.-P. Gallois, "Slicing communicating automata specifications: polynomial algorithms for model reduction," *Form. Asp. Comput.*, vol. 20, no. 6, pp. 563–595, Dec. 2008.

[8] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE '09, 2009, pp. 216–230.

[9] J. Wang, W. Dong, and Z.-C. Qi, "Slicing hierarchical automata for model checking uml statecharts," in *Proceedings of the 4th International Conference on Formal Engineering Methods*, 2002, pp. 435–446.

[10] V. Chimisliu and F. Wotawa, "Model based test case generation for distributed embedded systems," in *Industrial Technology (ICIT), 2012 IEEE International Conference on*, march 2012, pp. 656 –661.

[11] P. Samuel and R. Mall, "Slicing-based test case generation from uml activity diagrams," *SIGSOFT Softw. Eng. Notes*, vol. 34, no. 6, pp. 1–14, Dec. 2009.

[12] M. Weiglhofer and F. Wotawa, "Improving coverage based test purposes," in *Quality Software, 2009. QSIC '09. 9th International Conference on*, aug. 2009, pp. 219 –228.

[13] M. Bozga, J.-C. Fernandez, and L. Ghirvu, "Using static analysis to improve automatic test generation," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, pp. 142–152, 2003.

[14] D. A. da Silva and P. D. Machado, "Towards test purpose generation from ctl properties for reactive systems," *Electronic Notes in Theoretical Computer Science*, vol. 164, no. 4, pp. 29 – 40, 2006.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.