2013 13th International Conference on Quality Software

Improving Test Case Generation from UML Statecharts by using Control, Data and Communication Dependences

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*—Dependence relations have been used in slicing of programs in order to remove statements that do not influence certain criteria of interest. More recently, slicing has also been applied at the specification level in order to obtain a reduced model pertinent to the selected criteria. Such models have been used for different verification and validation activities. In this article we present an approach that uses control, data and communication dependences in order to enhance test purposes with refuse transitions. A test purpose represents an abstraction of the original model describing a scenario of interest, which should be tested. The refuse transitions are used during the test case generation process in order to limit the state space being searched. As automating test case generation activities is of great importance the generation of the test purposes and of the test cases consequently is fully automatic. We have evaluated the proposed approach on three models from industry and several others from literature. The obtained results indicate an improvement regarding computation time compared to test case generation using unmodified test purposes.**

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

I. Introduction

Due to the amount of resources consumed by testing activities in software development processes the automation of such tasks is very important. Automation of test case creation activities has received a lot of attention from the research community. Model based test case generation is a promising technique to help in the test case generation process. Model based test case generation assumes the availability of a model describing the desired behavior of the system under test (SUT).

The research in this area led to the creation of promising approaches and tools. In order to generate test cases, such a tool requires as input a formal description of the desired behavior of the SUT and some criteria to steer the test case generation. Such a criterium might be input from the user in the case of scenario based test case generation or some coverage metric the generated test cases need to achieve.

In our setting we model the desired behavior of the SUT by means of the modeling language UML [1]. The model itself describes a distributed system that uses asynchronous communication. Because UML lacks a formal semantic (it has however a formal syntax), we automatically extract a formal representation [2] in the form of a LOTOS [3] specification from the available model. LOTOS is a formal description technique developed within ISO for the formal specification of open distributed systems.

Once we obtain such a formal specification we are able to make use of already existing and mature tools (i.e., the CADP toolset [4]) in order to automatically perform verification and validation steps. In the current work, we are interested in the generation of conformance test cases by using the TGV [5] test-case generator from the CADP toolbox.

In order to generate test cases, besides the LOTOS specification TGV also needs a test purpose. A test purpose represents an abstraction of the original model describing a scenario of interest, which should be tested. These are used in order to focus the generation of test cases on particular aspects of the system. Test purposes are represented as labeled transition systems (LTS). They make use of 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 in a particular test-case generation process.

In our earlier work [6] we have introduced a method for automatically generating test cases aiming at structural coverage (state and transition coverage) of the model. We also showed how to semi-automatically generate test cases by making use of user provided annotations on the UML model. In that work the coverage generated test cases did not contain any refuse transitions. Therefore, the generation process was not as efficient as in the case when the user provides annotations that can be used as refuse transitions in the test purpose.

In the current article, we aim at using different dependence relations (control, data, and communication) in order to automatically identify parts of the models that can be safely omitted during the generation process. The dependence information is used to insert refuse transitions in the test purposes in order to focus test case generation and thus indirectly cut out transitions and states that are not of interest. Thus the specification is sliced during the generation process by not exploring its excluded parts. This is different to slicing approaches which eliminate transitions

978-0-7695-5039-8/13 $26.00 © 2013 IEEE
DOI 10.1109/QSIC.2013.48

IEEE computer society

(by merging their source and target states) on which the slicing criteria (transitions and/or variables) do not depend.

The rest of this paper is organized as follows. In Section II we shortly present the UML statecharts, LOTOS and the TGV testing approach. In Section III we describe the UML modeling assumptions and a running example, which is followed by Section IV where we define the used dependences. We furthermore introduce and discuss the algorithms for computing the dependences. In Section V we describe the test purpose generation process, and in Section VI we discuss first empirical results. Finally, we discuss related work in Section VII and conclude this paper in Section VIII.

II. Preliminaries

A. UML statecharts

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. Formally, we describe an UML statechart as a tuple $SC = (S, T, V, i_{ps0})$, where:

- $S \neq \emptyset$ is the set of states and pseudostates.
- $T \subseteq S \times L \times S$ is the set of transitions in the statechart. $L \subseteq E \times G \times A$ is the label of the transition. All components of a label are optional. E represents the set of triggers (events) that can fire the transitions, G the guards (conditions) that have to be true in order for the transitions to be fired and A - the actions that are to be performed when the transition is fired.
- V - the set of variables used in the statechart. These can be integer or boolean variables.
- i_{ps0} - the initial pseudostate that is the origin of the transition pointing to the initial state of the statechart.

More details regarding the syntax and semantics of statecharts can be found in the many books written about the language as well as in the UML standard [1].

B. LOTOS Introduction

LOTOS is a formal description technique developed within ISO for the formal specification of open distributed systems. It is composed of a process algebraic part and a data part based on the abstract data type language ACT ONE [7].

A LOTOS specification describes a system as a hierarchy of processes. Thus, a system can be modeled as a process that may contain several subprocesses. The LOTOS model of a system is viewed as a black box with a number of gates (interaction points) used for communication with its environment.

The behavior of a process is specified through behavior expressions composed of gate offerings and LOTOS operators. The operators are used to combine behavior expressions in order to form more complex expressions. As detailed information about the language can be found in [3], below we present only some of the LOTOS operators relevant to the used transformation:

- The sequentiality operator **";"**, called **action prefix** composes an action g with a behavior expression B. The expression describes a system that will initially accept action g behaving afterwards as B.
- **Choice "[ ]"** composes two alternative behaviors describing a system that offers these alternatives to the environment.
- The **full Synchronization: "||"** operator denotes the fact that the events which occur in either of the behavior expressions have to synchronize. The expression (a,b,X)||(a,b,Y) may engage in the sequence of events a,b... .
- The **interleaving** operator **"|||"** allows behaviors to unfold completely independently in parallel; the events from each behavior expression are interleaved. The behavior expression (a;b;c;P) ||| (x;y;T) includes the behaviors: a;x;y;b;c... and a;b;x;c;y... etc.
- **Partial Synchronization: "**$|[< gates >]|$**"** means that concurrent behaviors synchronize on the gates listed in the operator. The behavioral expression (a;b;c;P) $|[b]|$ (b;y;T) offers the behaviors a;b;y;c... , a;b;c;y... etc.

C. IOCO Relation and TGV

The Input Output Conformance theory (IOCO) introduced by Tretman [8] formalizes what it means that an implementation conforms to its specification. In IOCO the observations of a system during testing (also called traces) represent the visible behavior of a system. Informally, the IOCO relation states that an implementation I conforms to its specification S if after every trace, I exhibits the same outputs as S.

The IOCO relation represents the basis of the testing theory used by the test case generation tool TGV. For a formal and thorough description of this theory we refer the interested reader to [9]. TGV allows for the automatic generation of conformance test cases from formal specifications of reactive systems. By reactive we understand a software system which reacts to stimuli coming from its environment. Conformance testing aims at checking whether the visible behavior of a SUT is correct with respect to its specification.

In order to synthesize test cases, TGV (see Figure 1) requires the specification of the SUT that is usually given in a formal language like LOTOS, whose behavior can be expressed in terms of LTS. It also requires a test purpose defining the test scenario of interest, which is also represented as an LTS. A generated test case will aim at testing the functionality abstractly described by the test purpose. Another required input is the definition of the input/output alphabets of the specification representing the input and output actions of the system.

TGV makes use of enumerative techniques. An LTS representing the behavior described by the LOTOS specification is generated on the fly during the test case generation process.

The generated test cases are often described at some abstraction level (they can not be run directly against the SUT) so they must be translated into executable test cases

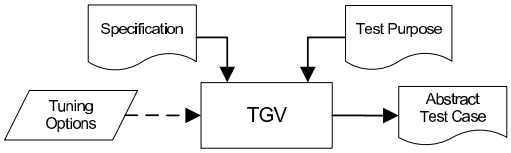

Figure 1. Functional View of TGV

e.g. TTCN [10] or any other format that can be executed against the SUT.

III. System Under Test

The behavior of the type of systems we consider is described by means of asynchronously communicating UML statecharts. By asynchronous we understand the fact that the events the statecharts use to communicate with each other are enqueued in a FIFO queue and consumed by the targeted statechart after it reaches a stable state. The events used in the communication with and within the system can also carry data values as event parameters.

The structure of the system is given in the form of a class diagram where each component of the system is represented by means of a class. The behavior of such a class is described by a statechart nested inside it.

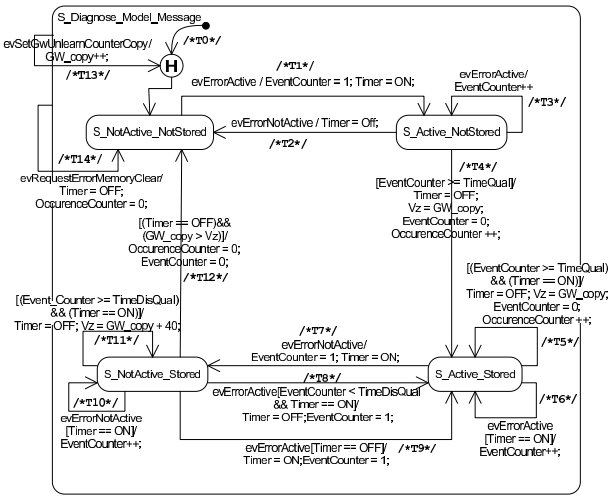

Figure 2. Statechart of Diagnosis Functionality

Our running example depicted in Figure 2, describes the diagnosis functionality of modern vehicles. Its 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. When such an error has been detected it is communicated to the diagnosis model.

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

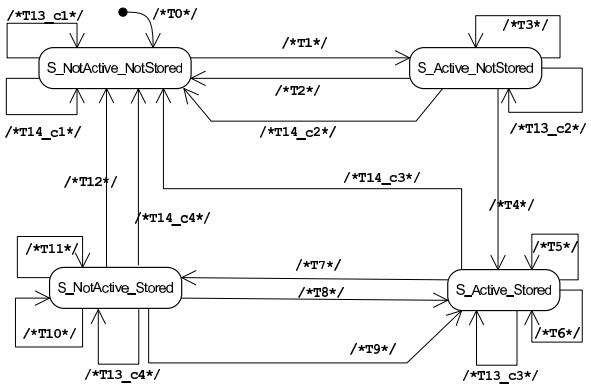

Figure 3. Flattened Representation of Diagnosis

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. The diagnosis module shall leave this state and move to the *S_NotActive_Stored* state only after receiving an *evErrorNotActive* event.

A. LOTOS Transformation

As already mentioned in Section I we use this model in order to automatically derive a LOTOS specification. The execution semantics of the UML model considered in the transformation are the ones defined by Harel [11].

Due to space considerations we are going to present only shortly the main points of the transformation of the UML model into LOTOS. For more details, the interested reader is referred to [2].

The first step of the transformation is represented by the flattening of the statechart in which the hierarchical structures and the pseudostates are removed. This step delivers a behavioral equivalent statechart described only in terms of simple states and transitions $SC = (S_s, T, V, i_{ps0})$. This is important for the current approach since the computation of the dependences is carried out on the flattened representation of the statechart. A detailed description of the flattening process can be found in [12].

Figure 3 presents the flattened representation of the diagnosis statechart from Figure 2. Due to readability reasons the labels of the transitions in Figure 3 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. In our running example (Figures 2 and 3) transition $T14$ generates the copies $T14_c1$, $T14_c2$, $T14_c3$ and $T14_c4$.

Each statechart in the system will be mapped to a LOTOS process while the variables used in the statecharts become LOTOS process parameters. These processes are then composed using the interleaving operator ("|||"). Each process will contain several sub processes used to represent the states in the statechart. Every subprocess offers choices between several behavioral expressions generated from the transitions of the state. Such an expression preserves the id, triggering event, guard and action of the transition. Once such a behavior expression has been triggered, all the components of the transition (event, guard, action - value assignments and/or generation of events to other models) are executed.

During the transformation process, LOTOS abstract data types are used in order to preserve the traceability between the components of the UML model and the generated LOTOS constructs. Thus in the generated specification we can still identify the statecharts, states, transitions, triggering events and variables used. This information is required in order to map the generated test cases back to the original UML model.

IV. 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 form sequential programs so control definitions have also been adapted for such formalisms. In [13] 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.

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 [13] (also acknowledged in [14]) 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 t_k of t_i (a maximal path) might not lead to $source(t_j)$ (t_j might not be executed). A transition t_k is a sibling of transition t_i if $source(t_k) = source(t_i)$. Considering the flattened representation (Figure 3) of the diagnosis model 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*.

1) Computing Control dependence: For computing the NTSCD relations we apply the algorithm in Figure 4 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 IV-A). In the algorithm we also make use of the function $maxPaths(t)$ (Equation 2) which provides all maximal paths starting with transition t.

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

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

The algorithm for computing the NTSCD (Figure 4) 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 t 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 t 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

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** $s_c \in \{s_c | \exists t_i \in \bigcap maxPaths(t) : source(t_i) = s_c\}$ **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 4. NTSCD Computation Algorithm

($outgTr(s_c)$) 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 siblings 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 [15] 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.

Due to the fact that 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: synchronization [16], communication dependence [14].

The one closest to what we need in our setting is the one given by [16] 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 automatons 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 [16] 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

Input: $T(M)$, CD, DD, $COMD$, TI
Output: $T_{IND}(t)$ - set of transition t does not depend on

1: **for all** $t \in T(M)$ **do**
2: $T_{CTRL}(t) \leftarrow T_{CTRL}(t) \cup t$
3: $T_{IND}(t) \leftarrow PI(t)$
4: $finished \leftarrow$ **true**
5: **repeat**
6: $finished =$ **true**
7: **for all** $t_i \in \{T(M) \setminus T_{CTRL}(t)\}$ **do**
8: **if** $controls(t_i, T_{CTRL}(t)) =$ **true then**
9: $finished \leftarrow$ **false**
10: $T_{CTRL}(t) \leftarrow T_{CTRL}(t) \cup t_i$
11: $T_{IND}(t) \leftarrow T_{IND}(D) \cup PI(t:i)$
12: **end if**
13: **end for**
14: **until** $finished$
15: $T_{IND}(t) \leftarrow T_{IND}(t) \setminus T_{CTRL}(t)$
16: **end for**

Figure 5. Independent Transitions Computation Algorithm

$COMD(t)$ 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 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 [14]). 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 $1 \leq n < k$: $t_n \xrightarrow{NTSCD} t_{n+1}$ or $t_n \xrightarrow{DD} t_{n+1}$ or $t_n \xrightarrow{COMD} t_{n+1}$.

In Figure 5 we present the algorithm for computing the independent transitions (T_{IND}) as well as the indirect dependences (T_{CTRL}) for each transition t in the model M (containing the communicating statecharts). The inputs for the algorithm are the set of transitions in the model $T(M)$ for which the dependence relations (CD, DD, $COMD$) and potential independents (TI) have been previously computed. The algorithm computes $T_{IND}(t)$ - the set of transitions that do not influence t for every transition in the model.

The function $controls(t_i, T_{CTRL}(t))$ (statement 8) returns true if at least a transition in T_{CTRL} is control, data or communication dependent on t_i. The algorithm iterates through the transitions of M and when it finds a transition t_i for which $controls(t_i, T_{CTRL}(t))$ returns *true* it updates the sets T_{CTRL} and T_{IND} accordingly. The algorithm is repeated until no more transitions are found such that $controls(t_i, T_{CTRL}(t)) =$ *true*. The set difference between T_{IND} and T_{CTRL} represents the transitions that do not influence (directly or indirectly) the transition t.

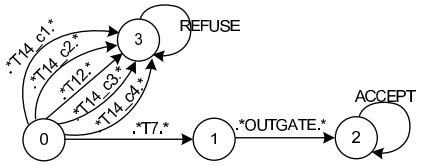

Figure 6. Test Purpose for T7

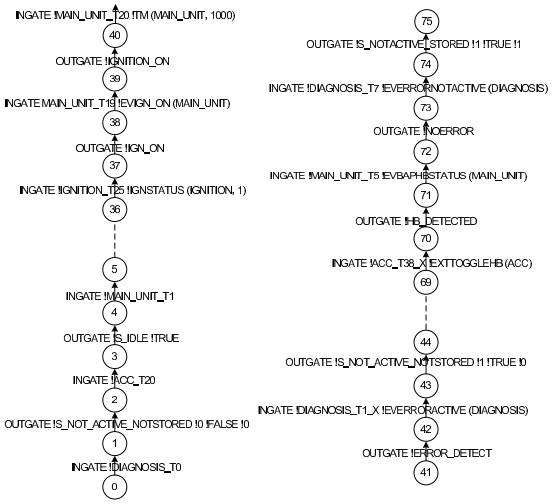

Figure 7. Test Case Covering Transition T7

V. Test Purpose Generation

A. Structural Coverage Aimed Test Case Generation

Since 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. By traceability we mean that in the LOTOS specification we are still able to identify the UML statechart elements such as transitions, states, variables and events. Thus, Figure 6 contains the LTS representation of a test purpose generated for covering transition *T7* in Figure 3.

The transitions leading to state 3 in Figure 6 are the independent transitions identified by applying the algorithm presented in Figure 5. Figure 7 contains the test case generated by using the afore mentioned test purpose.

In the test purpose definition, labels of transitions are denoted by strings that can also be stated 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) the labels of 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.

B. User Defined Test Purposes

Due to the fact that a test suite providing high coverage of the model does not guarantee that the system is well tested further mechanisms for the specification of test purposes are needed.

In [6] we also presented an approach for test case generation where test purposes were derived from annotations made on the model by the user. These annotations provided the order in which certain UML elements (states and/or transitions) needed to be visited. The amount of information in such a test purpose was always dependent on the user while the test generation tool had the task to compute the corresponding test case. The effort required to make such annotations also depends on the complexity of the model and that of the described test scenario.

Contrary to [6], the current approach tries to enrich the annotated test purposes with refuse transitions in order to reduce the state exploration of the input model

Technically the annotations are done using UML tags. Tags are pairs consisting of two elements - a name and a value. The name of the tag will represent the id of the scenario to cover with the generated test case. In order to specify a test scenario there are provided two types of annotations: inclusion and exclusion of UML statechart elements.

The semantic intended for the **included elements** is that they need to be part of the test case in the order provided by the annotation.

The **excluded items** are used to mark UML elements not pertinent to the current test scenario. We used these elements as refuse transitions in the derived test purposes.

Test purposes are automatically generated by using these annotations. The first step is moving the annotations on the states to all the transitions targeting them. Thus we obtain a specification of the test scenario only in terms of desired and excluded transitions.

We define a test purpose tp (Definition 7) as an ordered sequence of test purpose items tpi_j (Definition 6)

Definition 6: **(Test Purpose Item).** A **test purpose item** tpi is a tuple (IN, EX) where $IN \subset TR$ and $EX \subset TR$ are the included and excluded transitions of tpi. TR is the set of transitions in the model (from all statecharts).

Definition 7: **(Test Purpose).** A **test purpose** tp is a sequence $TPI = (tpi_0, tpi_1 ... tpi_n)$ of test purpose items where $\forall tpi_j, 0 <= j <= n : order(tpi_j) < order(tpi_{j+1})$.

We are going to use the Definitions 6 and 7 in order to describe the algorithm for generating test purposes from the user defined annotations. The basic idea for the algorithm

Input: $M, UDTP$
Output: $UDTP_{impr}$

1: **for all** $tp \in UDTP(M)$ **do**
2: $excludedT(tp) \leftarrow \bigcap\{EX(tpi_k) : tpi_k \in TPI(tp)\}$
3: $remove(excludedT, M)$
4: $T_{IND} \leftarrow computeIndeps(M)$
5: $communIndeps(tp) \leftarrow$
$\bigcap\{T_{IND}(tr) \mid tr \in IN(tpi_k) \wedge tpi_k \in TPI(tp)\}$
6: **for all** $tpi_k \in TPI(tp)$ **do**
7: $EX(tpi_k) \leftarrow EX(tpi_k) \cup communIndeps(tp)$
8: **end for**
9: $generateTestPurpose(tp)$
10: $UDTP_{impr} \leftarrow UDTP_{impr} \cup tp$
11: $add(excludedT, M)$
12: **end for**

Figure 8. Refuse Transitions Computation Algorithm

in Figure 8 is to identify the transitions that can be safely removed from the model and do not influence the included items of the user defined test purposes.

The inputs for the algorithm are the model M containing the different statecharts and a set of user defined test purposes $UDTP$. The output is the set of test purposes $UDTP_{impr}$ with potentially more excluded items. There is also the possibility that no further excluded transitions are added to the test purposes.

Making use of the user provided annotations we eliminate from the model (line 3) the transitions that have been marked as excluded for all the test purpose items tpi_k (line 2) of the current test purpose tp. At this point we have a new model on which we compute (line 4) the dependence relations described in the previous sections.

Since in a test purpose there are more than one included transitions only transitions that do not influence any of them (line 5) can be added as refuse transitions in the test purpose (line 7). At this point we go ahead and generate the test purpose (line 9).

Before moving on to the next test purpose we need to restore the model to its previous state (line 11).

VI. 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. We used the identified refuse transitions in order to improve a previously presented test case generation technique [6]. That technique was aimed at generating test cases for structural coverage of statecharts. The automatically generated test purposes did not contain any refuse states. Thus the present approach complements the test purposes with the refuse transitions identified by using the dependence relations.

Table I contains some statistical data regarding the models we used in our experiments. Thus the first column contains the name of the model while column $SCNo$ presents the

Table I
MODEL STATISTICS

$Model$	$SCNo$	$TrNo$	$StNo$	$TrNo_{flat}$	$StNo_{flat}$	$Proc_{LOTOS}$	LoC_{LOTOS}
Flasher	6	34	14	72	19	30	2800
Diagnosis	4	38	17	44	14	21	1800
KeylessEntry	3	35	22	43	13	19	1470
MicrowaveOven	2	34	12	37	10	15	1170
LoanApprovalWS	2	22	15	22	15	20	1210
ConferenceProtocol	3	41	18	41	18	24	1660
TelCtrlProtocol	2	55	26	70	21	26	2100

number of communicating statecharts of the model. Columns $TrNo$ and $StNo$ contain the number of transitions and states of the non flattened models. The number of transitions and states of the flattened version of the models can be found in columns $TrNo_{flat}$ and $StNo_{flat}$ respectively. Column $Proc_{LOTOS}$ contains the number of LOTOS processes derived from the model. The last column LoC_{LOTOS} contains an approximation of the number of lines of code in the LOTOS specification.

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 even if the flattened model contains them. Unfortunately, for now such situations are identified manually. Researching techniques to automatically find such situations is part of future work.

In Table II 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 [6]. 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 LTS level - the enumerated behavior of the specification) that is not considered during the test case generation process.

In some cases (*Diagnosis*, *MicrowaveOven* and *ConferenceProtocol*) 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.

We present the results of using the current technique with user defined test purposes in Table III. Here columns $DepT$ and $Time$ contain the time needed for the dependence computation and for generating the test cases in column TCs. The times in column $Time$ do not contain the ones for the computation of the dependence relations (column $DepT$).

For every model we defined a number of 15 test purposes and applied the algorithm in Figure 8 to try to complement them with refuse transitions. For most of the models (*KeylessEntry*, *MicrowaveOven*, *TelCtrlProtocol* and *LoanApprovalWS*) the generation time did not significantly improve compared with the old approach. This was to be expected since in one test purpose there are several transitions that have to be found and thus the chance of finding refuse transitions is also smaller.

In the case of the *Flasher* model, the current approach delivered better results. There were a number of four test purposes that account for the time difference between the two approaches. The size of the state space excluded through the added refuse transitions is one factor responsible for this time variation. Another influencing factor might be the algorithm TGV uses during the generation process and the order in which it visits transitions of the underlaying labeled transition system. The rest of the test purposes had comparable generation times.

Of course, when making use of dependence relations the test case generation results depend on the structure and size of the specification. However, in the worst case no refuse state might be introduced. The current approach is also influenced by the overhead imposed by computing the

Table II
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%
	AprI	72	70	70	-	58m10s	97%	100%
Diagnosis	Dep.	44	42	39	2s	1h20m	95%	97%
	AprI	44	42	38	-	1h43m50s	95%	97%
KeylessEntry	Dep.	43	39	39	2s	2m42s	91%	100%
	AprI	43	39	39	-	2m38s	91%	100%
MicrowaveOven	Dep.	37	37	37	1s	2m10s	100%	100%
	AprI	37	37	36	-	27m20s	97%	97%
LoanApprovalWS	Dep.	22	22	22	1s	1m9s	100%	100%
	AprI	22	22	22	-	1m20s	100%	100%
ConferenceProtocol	Dep.	41	41	41	3s	6m7s	100%	100%
	AprI	41	41	39	-	1h42m27s	95%	95%
TelCtrlProtocol	Dep.	65	65	65	2s	3m52s	100%	100%
	AprI	65	65	65	-	4m6s	100%	100%

Table III
USER DEFINED TEST PURPOSES RESULTS

Model	*Appr*	*TPs*	*TCs*	*DepT*	*Time*
Flasher	Dep.	15	15	1m52s	2m18s
	AprI	15	15	-	13m50s
Diagnosis	Dep.	15	15	9s	1m1s
	AprI	15	15	-	1m32s
KeylessEntry	Dep.	15	15	4s	52s
	AprI	15	15	-	52s
MicrowaveOven	Dep.	15	15	6s	55s
	AprI	15	15	-	57s
LoanApprovalWS	Dep.	15	15	5s	55s
	AprI	15	15	-	56s
ConferenceProtocol	Dep.	15	15	6s	12m21s
	AprI	15	15	-	15m5s
TelCtrlProtocol	Dep.	15	15	8s	51s
	AprI	15	15	-	55s

dependence relations. This overhead depends on the size and structure of the used model. For the used models this overhead was quite low.

VII. RELATED WORK

Slicing has been used in [17] 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.

Another approach [18] where test purposes are generated and extended with refuse states has also been proposed. There, 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.

A short version [19] of this paper will also appear. Additions in the current work are the user defined test purpose generation approach and the fact that more models have been used for the experimental results.

An approach using slicing for test case generation is [20], 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 criterium.

In [21] 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.

VIII. CONCLUSIONS AND FUTURE WORK

In this article we presented 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 [6] 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 and four from literature. 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. Such properties would help in deciding in which cases it makes sense to use the presented approach and in which to use other approaches.

Another direction of interest for future work is the investigation of the happens-before relation in case of com-

munication dependence. As also mentioned in [15] the communication dependence is not transitive thus reducing the precision of the obtained slices. The happens-before relation [22] 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.

Investigating techniques to automatically identify the unreachable transitions copies of the flattened model is also of interest for future work.

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] H. Garavel, R. Mateescu, F. Lang, and W. Serwe, "Cadp 2006: A toolbox for the construction and analysis of distributed processes," in *CAV*, 2007, pp. 158–163.

[5] 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.

[6] 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.

[7] J. de Meer, R. Roth, and S. Vuong, "Introduction to algebraic specifications based on the language ACT ONE," *Comput. Netw. ISDN Syst.*, vol. 23, pp. 363–392, 1992.

[8] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software - Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[9] J. R. Calamé, "Specification-based test generation with tgv," Centrum voor Wiskunde en Informatica, Technical Report SEN-R0508, May 2005.

[10] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, "An introduction to the testing and test control notation (ttcn-3)," *Computer Networks*, vol. 42, no. 3, pp. 375 – 403, 2003.

[11] D. Harel and H. Kugler, "The Rhapsody semantics of statecharts (or, on the executable core of the UML) - preliminary version," in *SoftSpez Final Report*, 2004, pp. 325–354.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] V. Chimisliu and F. Wotawa, "Using dependency relations to improve test case generation from UML statecharts," in *5th IEEE Int. Workshop on Software Test Automation*. IEEE Computer Society, 2013, p. to appear.

[20] 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.

[21] 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.

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