DTT-MAT: A Software Toolbox on Design-to-Test Approach for Testing of Embedded Programmable Controllers

Canlong Ma and Julien Provost

***Abstract*— This paper presents a continuation of previous results on a design-to-test (DTT) approach for black-box testing of embedded programmable controllers, where the specifications and implementations can be modeled as finite state machines [1]. The proposed approach checks the specification models and modify them in order to improve the testability of their physical implementation with limited design and testing overhead. This approach also guarantees by design that the behavior of the implementation remains unchanged during its normal execution (i.e. when not connected to a testbench). Based on proposed refinements of technical details and improvements of algorithms, a MATLAB toolbox 'DTT-MAT' has been developed.**

I. INTRODUCTION

Testing of industrial automation systems is facing challenges owing to new and diverse requirements from customers.

On the one side, in many applications, also in critical fields such as railway, power production and medical systems, automation systems are becoming more and more complex [2]. International safety standards such as [3] highly recommend the use and documentation of testing as a validation technique on top of formal verification methods, which brings a huge challenge to traditional testing approaches based on 'expert knowledge'.

On the other side, in some other fields, the life cycle of industrial products is also constantly shortening. As a consequence, the introduction of new products and modification of existing designs are carried out more frequently, which causes frequent changes in system specifications [4]. This also requires efficient system engineering approaches including powerful testing techniques.

Embedded programmable controllers such as ECU (Electronic Control Unit), PLC (Programmable Logic Controller) and SBC (Single-Board Computer) are widely used in various automation fields, also within safety critical systems such as automotive, railway, chemical industry, power production and medical systems. Compared to computers, which are global purpose, embedded programmable controllers are dedicated to a limited set of specific tasks.

To obtain a dependable and reliable system, requirement for testing tends to be automatic, efficient and economic. 'Automatic' means demanding as less 'expert knowledge' as possible, 'efficient' means generation and execution of test cases being simple and fast, 'economic' means costing as less testing overhead as possible.

Canlong Ma and Julien Provost are with Assistant Professorship for Safe Embedded Systems, Faculty of Mechanical Engineering, Technische Universität München, Garching bei München, Germany ma@ses.mw.tum.de; provost@ses.mw.tum.de

Recently, a design-to-test (DTT) approach, which aims at fulfilling these requirements, has been proposed in [1]. This approach can be applied to a broad field of automation systems where controllers receive inputs signals from multiple sources like sensors, internal buses and external networks.

In a traditional engineering process (Fig. 1, left V-model), testing is usually not concerned until the design phase is finished. In contrast, the proposed DTT approach takes testing performance into consideration during the design of specification (Fig. 1, right V-model). Before testing a programmable controller, the proposed DTT approach automatically updates the initial specification models in order to improve the testability of the final implementation.

In brief, the proposed DTT approach permits to save time during the test execution and guarantees better test coverage while paying limited design overhead.

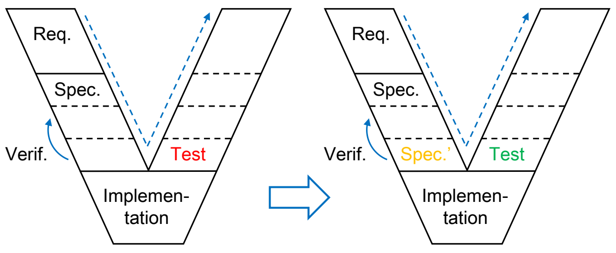


Fig. 1. V-model of the system engineering process

The contributions of this paper consist of: the refinement and optimization of the DTT algorithms, the development of the DTT-MAT toolbox, and its application to a realistic industrial case study.

The remainder of the paper is organized as follows: Section II presents the notations of Moore machines extended with Boolean signals, and black-box conformance testing for programmable controller. In section III, the testing issues encountered with embedded programmable controllers are reminded and the DTT methods are introduced with refined and optimized algorithms. The workflow of DTT-MAT and a case study are respectively illustrated in section IV and V. Discussion and perspectives are given in the last section.

II. BACKGROUND

A. Moore machine extended with Boolean signals

In this paper, the specification of a system is modeled as a set of Moore machines, and Boolean signals are used as inputs. In order to apply DTT-MAT, other kinds of formalisms could also be transformed into this notation. It has been proven that many other modeling languages can easily be translated into Moore machines, e.g., Mealy machines and Petri nets [5]. Besides, event inputs can also be expressed in equivalent signals. For example, a rising edge, as a typical event, is equivalent to a value change from '0' to '1' of a signal. Also, contrary to many event-based models, Moore machine extended with Boolean signals does not restrict signals to change only one at a time.

A Moore machine extended with Boolean signal inputs is defined by a 6-tuple $(S, s_{init}, I, O, \delta, \lambda)$, where:

- S is a finite set of states
- s_{init} is the initial state, $s_{init} \in S$
- I is a finite set of Boolean input signals
- O is a finite set of Boolean output signals
- $\delta : S \times 2^I \rightarrow S$ is the transition function, that maps the current state and the input signals to the next state
- $\lambda : S \rightarrow 2^O$ is the output function, that maps the states to their corresponding output signals

A transition guard is a Boolean expression that consists of input signal values and state variables. A transition is fired when its source state is active and its guard is evaluated as '1' (i.e. *True*). In this paper, the symbol $g(\delta)$ is used to denote the guard of a transition δ, and G to denote a set of transitions' guards.

In this paper, Moore machines are also represented in their graphical form. A simple example is given in Fig. 2. A state s is drawn as a circle (or a rounded rectangle) inside which a corresponding action $\lambda(s)$ is written (e.g. $A3$ in $S3$). A transition δ is represented by an oriented arc, upon which its guard $g(\delta)$ is placed (e.g. $\neg a \wedge b$ for the transition from $S1$ to $S2$).

A state can either have an externally observable action, e.g. $A3$ in S3 and $A4$ in $S4$, or no observable action, e.g. $\emptyset$ in $S1$ and $S5$. Besides, a state can also be given an internal state variable, e.g. $XS2$ in $S2$ and $XS6$ in $S6$. The internal state variables can also be used as Boolean transition guards, e.g. when the state $S2$ is activated, $XS2$ is assigned the value '1', then the transition from $S4$ to $S5$ can be fired.

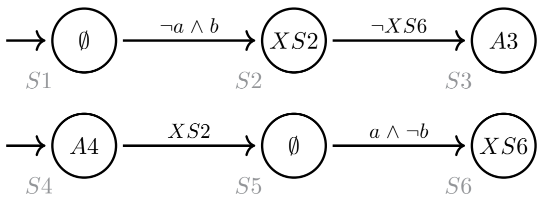


Fig. 2. A simple example of Moore machine with Boolean signals

The first step of the DTT approach is to composed in parallel all individual Moore machine models w.r.t stability research semantics. During the composition, a situation is transient if at least one transition of one of the Moore machines can be fired from this situation without change of the input signal values; a situation is stable when no more enabled transition can be fired from this situation for the current values of the input signals. The stability research semantics implies that the firing of transitions must continue until a stable situation is reached. For this purpose, DTT-MAT uses the Teloco algorithm proposed in [6].

Similarly to an individual Moore machine, a composed machine is defined by a 6-tuple $(L, l_{init}, I, O, \delta_L, \lambda_L)$, where:

- L is a finite set of locations
- l_{init} is the initial location, $l_{init} \in L$
- I is a finite set of Boolean input signals (same as for the individual models)
- O is a finite set of Boolean output signals (same as for the individual models)
- $\delta_L : L \times 2^I \rightarrow L$ is the evolution function, that maps the current location and the input signals to the next location
- $\lambda_L : L \rightarrow 2^O$ is the output function, that maps the locations to their corresponding output signals

A location represents a combination of states from the individual models. Besides, the symbols $g_E(\delta_L)$ and G_E are respectively used to denote the guard of an evolution δ_L and a set of evolutions' guards.

B. Black-box conformance testing

Black-box conformance testing is a testing strategy that doesn't peer at internal structure but only concentrates on the functional requirement, i.e. checking whether an implementation, seen as a black-box with inputs/outputs, behaves correctly with respect to its specification [7]. It is widely applied in late phases of testing, where the internal structures are not easily visible.

In this paper, the implementations under test (IUT) are physical embedded programmable controllers and the specifications are Moore machine models. The test objective is to perform a complete testing of the specification behavior (i.e. testing the behavior of the IUT for all combinations of input signals from all the states defined in the specification models).

A unit of a test sequence consists of three phases, each one containing a few steps [8]:

- Before testing the transition of interest:
  - bring the specification model and the IUT to a certain state by inputting a signal sequence (synchronizing or homing sequences)
- Testing the transition of interest:
  - apply the testing input signals to both the specification and the IUT
- After testing:
  - if needed, apply a distinguishing sequence to both the specification and the IUT
  - observe the emitted output signals by the specification model and the IUT
  - compare the results and continue to the next unit of the test sequence

The first and third phases constitute the testing overhead of a test sequence.

III. Testing issues & Refined DTT Approach

While many testing methods, e.g. [9], [10] and [11] aim at reaching a high coverage of all paths, the proposed DTT approach permits to reach a complete testing of the specification behavior w.r.t all the possible combinations of input signals from all states derived from specification models.

Several issues may occur during a complete testing, namely observability, controllability and single-input-change-testability (SIC-testability) issues.

A. Observability, Controllability and SIC-Testability Issues

The main objective of the proposed DTT approach is to slightly modify the specification models in order to automatically solve the three testing issues previously mentioned. A brief reminder of these issues is given below. Detailed information can be found in [1].

To realize the third phase of testing, the identification of the current state of the IUT, two ways can be considered: either by directly observing its output or by applying a distinguishing sequence. Generally speaking, the first way requires a strong hypothesis: every state must have a unique observable output action. However, this is not always fulfilled in real systems. The second way seems more general. However, it is also not always possible to find such sequences, and if there exist, they might be of exponential length to the number of states [8]. Obviously, for large scale systems, this will generate a huge testing overhead. This is what is referred to as the observability issue.

Similarly, during the first testing phase the specification and the IUT should be brought to a specific state. The controllability issue concerns how to rapidly control and bring the IUT from an arbitrary state to another known one. This issue can be solved by applying a homing or a synchronizing sequence. For complex systems, this process can also requires long sequences and thus generates lots of testing overhead.. This is what is referred to as the controllability issue.

In the second stage, i.e. during the actual test of the transition of interest, a set of input signals are read by the IUT. Because of cyclic input scanning, when several input signals change their values at the same time, the input values read by the IUT might deviate from the values supposed to be [12]. Thus, another transition than the one of interest could be fired. Such errors can be excluded if the test sequence contains only single-input-change test steps. This is what is referred to as the SIC-testability issue.

B. T-guard method

The proposed T-guard method (Alg. 1) permits to achieve a full SIC-testability by adding a minimum set of T-guards to some transitions in the models.

The example given Fig. 3 will be used to help illustrate the algorithm. The initial model is depicted in black; blue drawings correspond to the added T-guards. Since the example only contain one individual model, L and δ_L are equal to S and δ.

Algorithm 1: Pseudo-code of the T-guard method

```
   Input: $S$, $\delta$, $I$, $G_{E-NSIC}$
   Result: $\delta_{SIC}$
1  Initialization: $I_{NSIC} := \varnothing$; $T_{target} := \varnothing$;
2                  $S_T := \varnothing$; $\delta_{SIC} := \varnothing$;
3  begin
4      foreach $g_{E-NSIC} \in G_{E-NSIC}$ and $i \in I$ do
5          $I_{NSIC}$ += $\{i \mid i \wedge g_{E-NSIC} = g_{E-NSIC}\}$;
6          $I_{NSIC}$ += $\{\neg i \mid \neg i \wedge g_{E-NSIC} = g_{E-NSIC}\}$;
7      foreach $j \in I_{NSIC}$ do
8          $j := True$;
9          if $\bigwedge_{G_{E-NSIC}} g_{E-NSIC} \neq 0$ then
10             $T_{target}$ += $\{j\}$;
11             $j := False$;
12     foreach $(s \times g(\delta_i) \to s') \in \delta$ and $k \in T_{target}$ do
13         if $k \wedge g(\delta_i) = g(\delta_i)$ then
14             $S_T$ += $\{s\}$;
15     foreach $s_T \in S_T$ and $(s \times g(\delta_i) \to s') \in \delta$ do
16         if $s = s_T$ then
17             $g(\delta_i) := g(\delta_i) \wedge T\_guard$;
18             $\delta_{SIC}$ += $\{(s \times g(\delta_i) \to s')\}$;
```

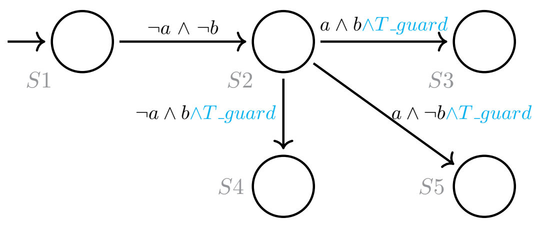

$S1$
$\neg a \wedge \neg b$
$S2$
$a \wedge b \wedge T\_guard$
$S3$
$\neg a \wedge b \wedge T\_guard$
$a \wedge \neg b \wedge T\_guard$
$S4$
$S5$

Fig. 3. A simple example of Moore machine updated with T-guards

Inputs of the algorithm are S, δ, I and G_{E-NSIC}. S, δ and I are respectively the union of the set of states, the union of the transition functions and the union of the input signals of all individual Moore machines. G_{E-NSIC} is a subset of evolutions' guards in the composed model that are non-SIC-testable, which is calculated by the tool Teloco [6] embedded in the DTT-MAT toolbox. Non-SIC-testable guards represents the input combinations that are not accessible by sole single input changes in existing evolutions. In the example, G_{E-NSIC} contains only one guard, i.e. $a \wedge b$ on the evolution from $S2$ to $S3$.

In Alg. 1, firstly the inputs that are involved in the non-SIC-testable evolution guards will be figured out (lines 4 to 6). T_{target} which is a minimum set of inputs that covers all g_{E-NSIC} is obtained by removing inputs iteratively (lines 7 to 11). In other words, all previous non-SIC-testable guards will be SIC-testable when these remaining inputs will be protected by T-guards. In the example, the T_{target} could be $\{a\}$ or $\{b\}$.

If a state has an outgoing transition whose guard is non-

SIC-testable, all outgoing transition guards from this state should be added with a T-guard (lines 12 to 14 and lines 15 to 18). In the example, all the transitions outgoing from $S2$ will be added with T-guards.

The result of this algorithm is δ_{SIC}, a transition function which contains only transitions with updated guards, i.e. added with T-guards. All the previous non-SIC-testable parts of the locations can now protected by T-guards, so any outgoing transition requiring a multiple-input-change test step to be fired can be temporary frozen by the T-guards, i.e. by setting the value of the input signal T_guard to the value '0'. Now the system can be completely tested without errors due to asynchronism between input signals.

Once the conformance testing completed, and before running the IUT in its normal mode, the input signal T_guard will be connected to the logic 1 level (3.3V, 5V or 24V depending on the implementation architecture) so that it will not affect the original transition guards ($g \wedge 1 = g$).

C. O-action method

Alg. 2 presents the method of how to achieve a full observability with a limited design overhead by adding O-actions into some of the states.

Algorithm 2: Pseudo-code of the O-action method

```
   Input: $L$, $S$, $\lambda$, $\lambda_L$, $\#_{indi}$
   Result: $\lambda_{OBS}$
1  Initialization: $S_{NOBS} := \varnothing$;  $O_{OBS} := \varnothing$;
2  begin
3      foreach $(l_i, l_j) \in L^2, l_i \neq l_j$ do
4          if $\lambda_L(l_i) = \lambda_L(l_j)$ then
5              for $n = 1 \colon \#_{indi}$ do
6                  if $\lambda_n(s_i) = \lambda_n(s_j)$ , $s_i \in l_i, s_j \in l_j$ then
7                      $S_{NOBS}$ += $\{s_i, s_j\}$;
8      $\#_{OA} := \lceil \log_2 (|S_{NOBS}|) \rceil$; // number of O-actions
9      $OA := \left[ oa_1, oa_2, \cdots, oa_{\#_{OA}} \right] \in O^{\#_{OA}}$;
10     // $oa$ is a single O-action, OA is a list of $oa$
11     foreach $s \in S_{NOBS}$ do
12         $\lambda(s) := \lambda(s) \wedge minterm(OA)$;
13         // where $minterm(OA)$ returns a unique
           combination of $OA$ elements
14         $\lambda_{OBS}$ += $\{\lambda(s)\}$;
```

In Alg. 2, among other common inputs similar to those of Alg. 1, $\#_{indi}$ is the number of individual models.

First of all, output actions of all the locations will be examined (lines 3 to 4). If at least a pair of locations share the same actions, the individual states inside the locations will be further analyzed (lines 5 to 6). If two different states in the same individual model have the same action, then they are identified as the cause of non-observability of those locations. These states will be collected in S_{NOBS} (line 7).

Since each O-action can be set the value '0' and '1', a list of n O-actions can be used to represent 2^n different outputs. Thus, a minimum set of O-actions can be obtained by applying the formula $\#_{OA} := \lceil \log_2 (|S_{NOBS}|) \rceil$.

A unique O-action combination will be given to each state from S_{NOBS} (lines 11 to 14). As a result, the previous non-observable locations will become fully observable.

It is also worth mentioning that O-actions are only observable output signals. They do not affect the rest of the system behavior throughout testing and normal executions.

D. C-guard method

The goal of the C-guard method is to solve the controllability issue, i.e., to shorten the distance between locations during testing. It is realized by adding a set of C-guard transitions to the models.

In some models, some states may not reachable from some other states, e.g. in the example in Fig. 4, once $\delta 02$, $\delta 24$ have been taken, $S1$ cannot be reached any more, so $\delta 01, \delta 13$ cannot be tested. Now, C-guards make it possible to completely test all transitions.

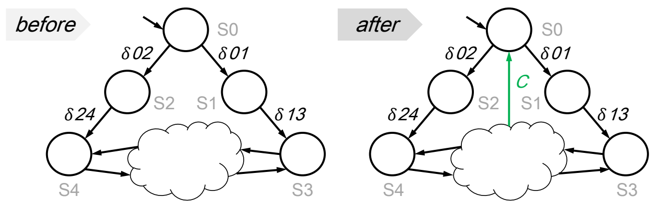


Fig. 4. C-guard in testing transitions between unreachable states

In this paper, the introduction of C-guard method focuses on the global algorithm (Alg. 3), while specific functions have already been explained in details in [1].

$Dist_L$ is a path cost matrix for all couples of locations. During initialization (lines 3 to 9), if there is a direct evolution from one location to another, the path cost for the couple of locations will be set to 1, if not, then to ∞. The Floyd-Warshall algorithm is then applied to calculate indirect path costs between all couples of location (line 10).

After that, the maximum path cost will be compared to $Limit_C$, the expected path cost limit. If the maximum path cost exceeds this limit, a set of new evolutions δ_{L-C} will be built (lines 11 to 13). Details of the method $EvoCalc(Dist_L, Limit_C)$ are given in [1].

Based on the result of δ_{L-C}, a minimum set of transitions with associated C-guards δ_C for individual models, will be calculated (lines 14 to 21). In the example in Fig. 5, if the path cost from $S1$ to $S2$ exceeds $Limit_C$, then a

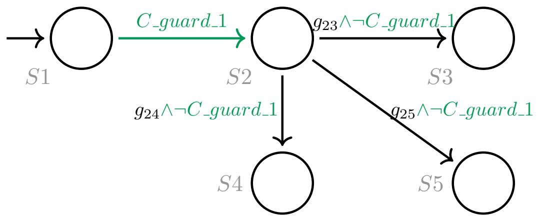

C_guard_1
$g_{23} \wedge \neg C\_guard\_1$
$g_{24} \wedge \neg C\_guard\_1$
$g_{25} \wedge \neg C\_guard\_1$
$S1$
$S2$
$S3$
$S4$
$S5$

Fig. 5. A simple example of Moore machine updated with C-guards

new transition from $S1$ to $S2$ will be built with the guard C_guard_1.

It's important to mention that for stability reason the negation of the C-guards will added to the guards of the destination state (lines 27 to 29). In the example in Fig. 5, the guards on all the outgoing transitions from $S2$ will be added with $\neg C\_guard\_1$.

Similarly to T-Guards, once the conformance testing completed, the input signals C_guard_i will be connected to the logic 0 level (0V or below 1V for most of the architectures) so that they will never enable the firing of these transitions ($g \wedge 0 = 0$).

Algorithm 3: Pseudo-code of the C-guard method

```
   Input: $L$, $\delta_L$, $\delta$, $Limit_C$, $\#_{indi}$
   Result: $\delta_C$
1  Initialization: $\delta_{L-C} := \varnothing$;  $\delta_C := \varnothing$;
2  begin
3      foreach $(l_i, l_j) \in L^2$ do
4          if $l_i = l_j$ then
5              $Dist_L(l_i, l_j) := 0$ ;
6          else if $\exists\ (l_i \times g(\delta_i) \to l_j) \in \delta_L$ then
7              $Dist_L(l_i, l_j) := 1$ ;
8          else
9              $Dist_L(p, q) := \infty$ ;
10     $Dist_L := Floyd - Warshall(Dist_L)$;
11     while $\max(Dist_L) > Limit_C$ do
12         $\left(Dist_L, \delta_{L-new}\right) :=$
           $EvoCalc(Dist_L, Limit_C)$;
13         $\delta_{L-C} \mathrel{+}= \ \delta_{L-new}$;
14     foreach $(l_{src} \times g(\delta_{Li}) \to l_{des}) \in \delta_{L-C}$ do
15         for $n = 1 \colon \#_{indi}$ do
16             if $\nexists(s_{src} \times g(\delta_i) \to s_{des}) \in \delta, s_{src} \in$
               $l_{src}, s_{des} \in l_{des}$ then
17                 $\delta_C \mathrel{+}= \ \{(s_{src} \times 1 \to s_{des})\}$;
18     $\#_C := |\delta_C|$; // number of C-guards
19     $C := \left\{c_1, c_2, \cdots, c_{\#_C}\right\}$; // set of C-guards
20     foreach $\delta_{Ci} \in \delta_C$ do
21         $g(\delta_{Ci}) := C(i)$; // $i$ is the index of $\delta_{Ci}$ in $\delta_C$;
22         foreach $\delta_j \in \delta \mid$
           $\exists g(\delta_j), \delta(dest(\delta_{Ci}), g(\delta_j)) \neq \varnothing$ do
23             $g(\delta_j) := g(\delta_j) \wedge \neg C(i)$;
```

IV. DTT-MAT: MATLAB Toolbox for DTT Approach

A toolbox DTT-MAT has been developed to realize the proposed DTT approach. It is available from our homepage: www.ses.mw.tum.de.

A. Workflow of DTT-MAT

The workflow of DTT-MAT is briefly depicted in Fig. 6.

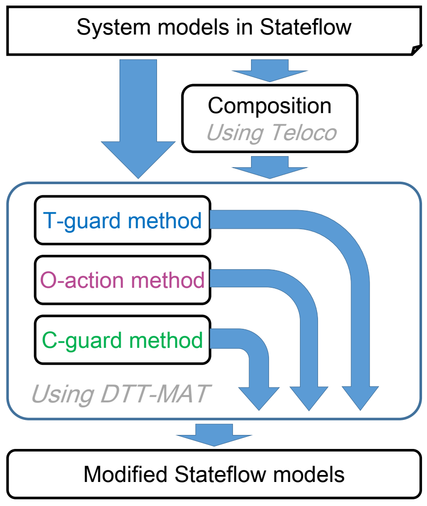


Fig. 6. Workflow of DTT-MAT

To use DTT-MAT, the user should firstly model the system in MATLAB Stateflow. Usually, a complex system is split into several individual models for reasons of simplicity.

Stateflow models are first transformed into Moore machines, which will next be read by Teloco [6]. Then, all the individual models will be composed with Teloco, and SIC-testability results will also be generated.

The Stateflow models and the composed model from Teloco will then be analyzed by the proposed DTT methods implemented in MATLAB. The three DTT methods can be executed one after another, which is recommended when modifying a new system; but is also possible to run them separately, e.g. to compare the impact of the value of $Limit_C$

Based on the results from the DTT methods, the Stateflow models will be automatically updated with T-guards, O-actions and C-guards, in order to fulfill the observability, controllability and SIC-testability requirements.

Additionally, automatic code generation has been implemented for PLC, using IEC 61131-3 Structured Text format. Next step for the development of DTT-MAT will be automatic generation of C code from updated specification models.

B. Limitations for applicable Stateflow models

MATLAB Stateflow offers rich possibilities to build models. For example, signals and events are both accepted as inputs, actions can be linked to states, transitions or transition conditions [13].

However, only Moore machine models can be handled with the current version of Teloco. Besides, only Boolean signals are accepted as valid inputs for the current version of DTT-MAT. Detailed instructions as well as some examples are available together with the toolbox.

V. Benchmark Case Study

In this paper, a case study (Fig. 7) slightly adapted from [4], is used to illustrate the DTT approach and the DTT-MAT toolbox. For more information upon the application of the DTT approach, [14] provides two other case studies on industrial applications.

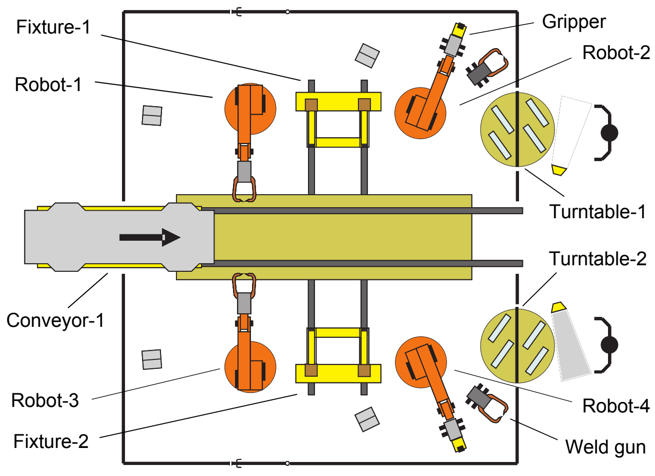


Fig. 7. Case study: a welding and material handling cell

A. Description of system

This case study contains nine machines: four robots, two fixtures, two turntables and a conveyor.

This cell does welding tasks in three phases. At the beginning, *Conveyor-1* delivers a car body into the cell. *Robot-1* begins to weld the parts, which are loaded by previous systems. This is the first weld job (J1). Meanwhile, *Robot-2* picks plates from *Turntable-1* and places them in *Fixture-1*. *Turntable-1* turns when two plates have been taken. In the second job (J2), *Robot-1* and *Robot-2* work together to weld the plates held by fixture to the car body. After they finish J2, *Fixture-1* moves away from its workstation, to enable *Robot-1* and *Robot-2* weld the parts which were blocked by *Fixture-1*. This is the third Job. On the opposite side, the same work will be executed by *Robot-3*, *Robot-4*, *Turntable-2* and *Fixture-2*. As soon as the weld jobs are completed, the robots also move away from their workstation. *Conveyor-1* delivers then the car body out of the cell. Afterwards, the robots and fixtures move back to their workstations, making the cell ready for next round.

A few coordinators are set up to control the correct operation of all individual machines and robots. The complete system can thus be modeled by 12 individual Stateflow models, which result in a composed model containing 792 stable locations and 93,587 evolutions. The system has 34 Boolean inputs and 33 Boolean outputs. A selection of inputs and outputs for *Robot-2* is given in Fig. 8.

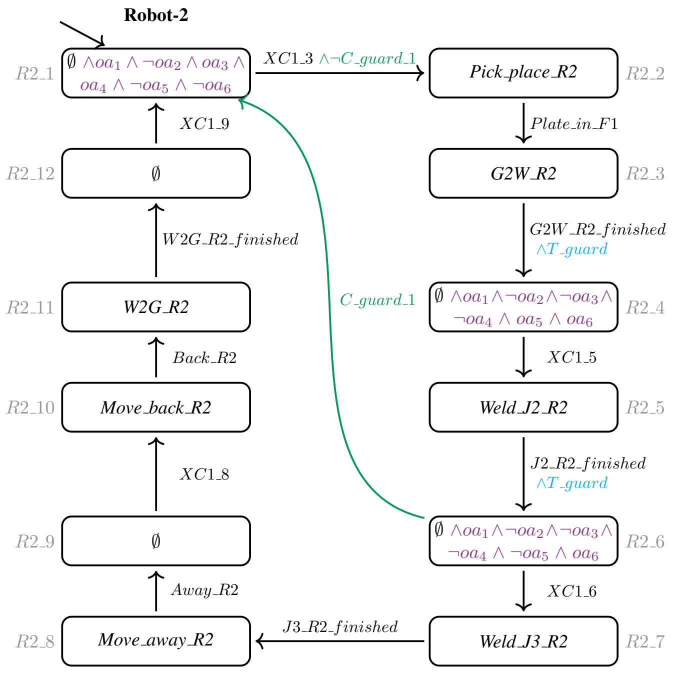
The model for *Robot-2* is selected as an illustrative example for this section (Fig. 9). The initial model is depicted in black; blue, green and purple drawings and text correspond to the elements added by DTT-MAT.

B. Testing issues

When observing the individual models, it can easily be found that some states have the same output actions. For example, in *Robot-2*, the states *R2-1*, *R2-4*, *R-6*, *R-9*, *R-12*

Machine	**Input**	**Output**
Robot-2 (R2)	**Plate_in_F1**	**Pick_place_R2**
	detect if plate is placed in Fixture-1	pick the plate and place it in Fixture-1
	G2W_R2_finished	**G2W_R2**
	detect if R2 finishes tool change G2W	change tool from gripper to weld gun
	W2G_R2_finished	**W2G_R2**
	detect if R2 finishes tool change W2G	change tool from weld gun to gripper
	J2_R2_finished	**Weld_J2_R2**
	detect if R2 finishes Job2	do the Weld-Job2
	J3_R2_finished	**Weld_J3_R2**
	detect if R2 finishes Job3	do the Weld-Job3
	Away_R2	**Move_away_R2**
	detect if R2 is away from workstation	move away from workstation
	Back_R2	**Move_back_R2**
	detect if R2 is back to workstation	move back to workstation

Fig. 8. Boolean inputs and outputs for the model *Robot-2*



Robot-2
R2_1
$\emptyset \wedge oa_1 \wedge \neg oa_2 \wedge oa_3 \wedge oa_4 \wedge \neg oa_5 \wedge \neg oa_6$
$XC1\_3 \wedge \neg C\_guard\_1$
Pick_place_R2
R2_2
Plate_in_F1
$XC1\_9$
R2_12
$\emptyset$
G2W_R2
R2_3
W2G_R2_finished
G2W_R2_finished
$\wedge T\_guard$
R2_11
W2G_R2
C_guard_1
$\emptyset \wedge oa_1 \wedge \neg oa_2 \wedge \neg oa_3 \wedge \neg oa_4 \wedge oa_5 \wedge oa_6$
R2_4
Back_R2
$XC1\_5$
R2_10
Move_back_R2
Weld_J2_R2
R2_5
$XC1\_8$
J2_R2_finished
$\wedge T\_guard$
R2_9
$\emptyset$
$\emptyset \wedge oa_1 \wedge \neg oa_2 \wedge \neg oa_3 \wedge \neg oa_4 \wedge \neg oa_5 \wedge oa_6$
R2_6
Away_R2
$XC1\_6$
R2_8
Move_away_R2
J3_R2_finished
Weld_J3_R2
R2_7

Fig. 9. A Moore machine model for Robot-2

don't have any observable action. This implies that, after composition, it is possible that some locations have the same actions, which leads to the observability issue.

Even though the composed model contains 93,587 stable evolutions, not all locations can directly be reached from other locations. Thus, the distance from some states to other states could be very long, i.e., this system might have the controllability issue.

C. Testability after applying DTT-MAT

With the help of DTT-MAT, quantitative results can be automatically obtained for the testing issues.

Out of 792 locations, 777 of them are not fully SIC-testable. Applying T-guard method, one feasible solution is found, 12 Boolean inputs out of 34 are involved in non-SIC-testable transitions guards. After updating the original specification models with 14 T-guards, the composed machine reaches a full SIC-testability. Two of the T-guards have been drawn in blue in Fig. 9.

Then, 537 locations have the same output action with at least one of the other locations. With traditional methods, a distinguishing sequence may be 537 steps long. However, analysis with the O-action method shows that the observability issue of 537 locations was caused by 33 states in the individual models. After adding 6 O-actions (drawn in purple Fig. 9), the issue can easily be solved.

Finally, according to the C-guard method results, some locations are not reachable from some other locations.

After adding 14 C-guards on individual models (one of them have been drawn in green in Fig. 9), any location can be reached within maximum 4 steps from any other location in the composed model.

D. *Example of executable code generation*

The automatically generated ST code for PLC from the initial models of the case study contains 350 lines. Adding the 14 T-guards only increase the code length by 1 line (declaration of the T_guard variable). Also, because T-guards are only added to existing transitions guards instead of creating new transition, 14 lines of code are modified by adding 'AND T_Guard' to existing guards.

Adding the 6 O-actions increases the code length by 12 lines: 6 lines to declare outputs variables and 6 lines to assign the conditions when the O-actions are activated. An example of ST code for O-action assignment is as follows:

- $oa2$:= *XR1_3 OR XR1_1 OR XR4_1 OR XR4_6 OR XR3_1 OR XR3_3 OR XF1_1 OR XF2_1 OR XT1_2 OR XT1_1 OR XT2_2 OR XT2_1 OR XC1_1 OR XC1_2 OR XC2_1 OR XC2_2 OR XC3_1 ;*

Adding the 14 C-guards increases the code length by 28 lines: 14 lines to declare inputs variables and 14 lines to create new transitions with the C-guards. An example of ST code for a new transition with C-guard is as follows:

- $tR21$:= *XR2_6 AND C_guard_1;*

Thus, concerning the design overhead added by the DTT methods, it can be positively concluded that the increase of the code length is linear in terms of added lines of code to the number of inserted O-actions and C-guards, and is linear in terms of modified lines of code to the number of T-guards.

VI. Discussion and perspectives

This paper has presented improved algorithms for the design-to-test approach for testing embedded programmable controllers, and *DTT-MAT*, a MATLAB toolbox supporting this approach.

The proposed DTT approach aims at improving the testability and reducing the testing overhead with limited design overhead.

Specifically, DTT-MAT accepts system specifications modeled as Moore machines with Boolean signal inputs in MATLAB Stateflow. By running the T-guard, O-action and C-guard methods, the specification models are modified to meet the requirements of full SIC-testability, full observability and better controllability. A realistic industrial case study is used to illustrate the application of DTT-MAT.

A key point of the proposed DTT method to remind is that, during normal execution, all T-guards and C-guards can be inhibited (by connecting them to the logic 1 and 0 levels, respectively), and all O-actions are only additional output signals that can be ignored. Thus, none of the added T-guards, C-guards and O-actions is affecting the behavior of the system in its normal mode. Also, all these T-guards, O-actions and C-guards could be use again during maintenance and inspection to assist in the identification of problems.

Future work is aiming at extending this DTT approach to more general applications, e.g. specification models with analog signals.

References

[1] C. Ma and J. Provost, "Design-to-Test Approach for Black-Box Testing of Programmable Controllers," in *IEEE Int. Conf. on Automation Science and Engineering (CASE), 2015*, 2015, pp. 1018–1024.

[2] B. Vogel-Heuser, C. Diedrich, A. Fay, S. Jeschke, S. Kowalewski, M. Wollschlaeger, and P. Göhner, "Challenges for Software Engineering in Automation," *Journal of Software Engineering and Applications*, vol. 07, no. 05, pp. 440–451, 2014.

[3] IEC61508, *Functional safety of electrical / electronic / programmable electronic safety-related systems*, 2nd ed. International Electrotechnical Commission, 2010.

[4] J. Richardsson and M. Fabian, "Modeling the control of a flexible manufacturing cell for automatic verification and control program generation," *Int. Journal of Flexible Manufacturing Systems*, vol. 18, no. 3, pp. 191–208, 2007.

[5] C. K. Chang and H. Huang, "On transforming Petri net model to Moore machine," in *14th Annual Int. Computer Software and Applications Conf.*, no. 052. Chicago: IEEE, 1990, pp. 267—-272.

[6] J. Provost, J. M. Roussel, and J. M. Faure, "Translating Grafcet specifications into Mealy machines for conformance test purposes," *Control Engineering Practice*, vol. 19, no. 9, pp. 947–957, 2011.

[7] R. S. Pressman, *Software Engineering A Practitioner's Approach Seventh Edition*. MC Graw Hill, 2010.

[8] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines – a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[9] a. S. Kalaji, R. M. Hierons, and S. Swift, "An Integrated Search-Based Approach for Automatic Testing from Extended Finite State Machine (EFSM) Models," *Information and Software Technology*, vol. 53, no. 12, pp. 1297—-1318, 2011.

[10] A. Denise, M. C. Gaudel, S. D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet, "Coverage-biased random exploration of large models and application to testing," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 1, pp. 73–93, 2012.

[11] T. Shu, T. Ye, X. Yin, and J. Xia, "A Test Generation Method for EFSM-based Protocols Using the Transitions Feasibility Estimation," *International Journal of Control and Automation*, vol. 9, no. 5, pp. 207–218, 2016.

[12] J. Provost, J.-M. Roussel, and J.-M. Faure, "Generation of Single Input Change Test Sequences for Conformance Test of Programmable Logic Controllers," *IEEE Trans.on Ind. Inform.*, vol. 10, pp. 1696–1704, 2014.

[13] Mathworks, "Stateflow User's Guide," Tech. Rep., 2015. [Online]. Available: http://nl.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf

[14] C. Ma and J. Provost, "Design-to-test : an approach to enhance testability of programmable controllers for critical systems – two case studies," in *European Conference on Safety and Reliability - ESREL 2016*, Glasgow, Scotland, 2016, pp. Sept, 2016.