Test Adapter Generation Based on Assume/Guarantee Contracts for Verification of Cyber-Physical Systems

Jishu Guin[a], Jüri Vain[b] and Leonidas Tsiopoulos[c]
Department of Software Science, Tallinn University of Technology, Ehitajate tee 5, Tallinn, Estonia
{*jishu.guin, juri.vain, leonidas.tsiopoulos*}*@taltech.ee*

Keywords: Model-Based Testing, Provably Correct Test Development, Test Adapters, Assume/Guarantee Contracts, Model-Checking, Uppaal Timed Automata, Code Generation.

Abstract: Test adapter generation forms an essential but often least automatized part in the Model-Based Testing (MBT). The difficulty of adapter generation is due to ambiguous or loosely defined mapping between the executable test interface and that of abstracted in the test model. The novel method exposed in this work uses saturated Assume/Guarantee (A/G) contracts to specify test interfaces and to generate adapters from those. As a contribution, firstly, we define a generic saturated A/G contract template that supports the uniform approach to specification and verification of test configuration components. Secondly, we demonstrate how the adapter component model is derived by refining the test interface contracts and its correctness is verified. Finally, the adapter code is generated from the verified model as a set of abstract to concrete and concrete to abstract symbol transformers. The approach is exemplified and validated on a real climate control system example.

1 INTRODUCTION

Model-based testing (MBT) (Utting et al., 2012) provides the formal rigor required to assure safety, resilience and functional qualities of complex cyber-physical systems (CPS). Though most of industry strength testing tools provide some test automation support [1], one of the distinguishing features of MBT is the effort and expertise needed to map the real SUT to its abstract representation and to specify test I/O. Guin et al. (2022) showed that contracts can enhance the testing process in large by providing structured and verifiable specifications of test configuration. The current work is an extension of this demonstrating that contracts are applicable likewise to test adapter generation enabling almost complete test development and deployment automation.

As a first contribution, we employ a generic specification template of the Assume/Guarantee (A/G) contracts enriched by timing conditions and formalized in Timed Computation Tree Logic (TCTL) (Alur et al., 1990) and show that it is expressive enough to incorporate information for adapter code generation.

Relying on the unified contract format, we demonstrate that adapter functionality, considered as a set of symbol transformations, further called transformers, allows conjoining transformers contracts to the adapter contract. This enables contracts non-interference verification (Bensalem et al., 2010).

The specifics of adapter contracts lies in its purpose to bridge the different abstraction levels of the model and executable test interface. As a third contribution, an algorithm to generate an adapter model in Uppaal Timed Automata (UTA) formalism from its contract is presented. It allows the integrity and consistency checking of the test configuration refined with deployment details and extracting code from it.

2 RELATED WORKS

The idea of using the principles of Design-by-Contract in software testing is not new. Several papers ((Ciupa and Leitner, 2005; Milicevic et al., 2007)), influenced by the Eiffel programming language (Meyer, 1988) use pre- and post-conditions in code as specifications for test oracles. The level of specification abstraction was raised, e.g., in UML-based Visual Contracts (Güldali, 2014). It was shown that pre/post-condition based paradigm is well suited for off-line testing of data-intensive systems where data coverage

[a] https://orcid.org/0000-0001-5522-2194
[b] https://orcid.org/0000-0002-0700-7972
[c] https://orcid.org/0000-0002-3994-3810
[1] https://testsigma.com/blog/model-based-testing-tools/

Guin, J., Vain, J. and Tsiopoulos, L.
Test Adapter Generation Based on Assume/Guarantee Contracts for Verification of Cyber-Physical Systems.
DOI: 10.5220/0013322100003896

In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 297-304
ISBN: 978-989-758-729-0; ISSN: 2184-4348

is the main concern (Güldali, 2014; Barr et al., 2015).

Our focus is online testing where the models of the SUT and test oracle are developed in a correct-by-construction manner from system requirements agnostic to specific software code. One of the pioneering works by (Blundell et al., 2005) in this direction incorporates system design level A/G reasoning theory for safety-critical system MBT. However, when using symbolic witness traces generated by model checkers, an extensive manual labor is required to translate the symbolic traces to executable test data. In recent years few works were published incorporating state machine formalisms for encoding contracts which are manually inserted around SUT program statements cf. (Boudhiba et al., 2015; Xu et al., 2016). However, automatic generation of test adapters and merging them with the rest of the test model needed for online MBT, especially for systems with non-deterministic behavior, has deserved surprisingly little attention. It is rather common to leave the adapter implementation to test engineer's responsibility which is time consuming and error prone. The main motivator of this work is to fill this gap by an automated procedure.

3 PRELIMINARIES

3.1 Model-Based Testing

MBT provides the test verdict based on the conformance relation between the observable I/O behavior of the SUT and its model. The test fails when the SUT and model behaviors do not match. The test inputs are generated using assumptions on SUT environment behavior that are grouped into test cases. The mainstream MBT methods focus on I/O conformance (IOCO) testing. However, since CPS requirements generally refer also to timing constraints, the stronger conformance relation RTIOCO that covers also timing (Larsen et al., 2005), is applied in this study. Model-checking (Alur et al., 1990) is used in MBT to generate test cases where the coverage criteria are expressed in terms of properties verified and their witness traces used as symbolic test sequences. The test coverage properties are extracted from system requirements and expressed as observer automata or in logics, e.g., temporal logic CTL (Wijesekera et al., 2007), Quantum Hoare logic (Kumar, 2023), etc. Executable test cases are extracted from the symbolic traces and instantiated with test data by means of test adapters. Alternatively, the symbolic traces are compiled to test scripting language, e.g., TTCN-3.

The testing process according to standard ANSI/IEEE 829-1983 comprises test planning, design, execution, and results analysis. Provably correct MBT means that transit to next phase is done only when the correctness of current step is verified. While the correctness properties of the development phases have been introduced in (Vain et al., 2017), in this paper we focus on test deployment and specifically on the provably correct adapter generation.

3.2 Uppaal Timed Automata

The real-time requirements of CPS impel the use of a formalism adjusted for real-time systems. UTA (Behrmann et al., 2004) is defined as a closed network of parameterized extended timed automata that when parameters are instantiated are called *processes*. The processes are composed by synchronous parallel composition. The nodes of the automata graph are called *locations* and directed arcs between locations are called *edges*. The *state* of an automaton consists of its current location and valuation of variables, including clocks. The synchronization of processes is defined using *channels*.

Formally, an UTA is a tuple (L, E, V, CL, $Init$, Inv, T_L), where L is a finite set of locations , E is the set of edges defined by $E \subseteq L \times G(CL,V) \times Sync \times Act \times L$, where $G(CL,V)$ is the conjunction of transition enabling conditions, $Sync$ is a set of channels and Act is a set of assignments with integer and boolean expressions and clock resets. V denotes the set of variables of boolean and integer type and arrays of those. CL denotes the set of real-valued clocks ($CL \cap V = \emptyset$). $Init \subseteq Act$ is a set of initializing assignments to variables and clocks. $Inv : L \to I(CL,V)$ maps locations to the set of invariants. $T_L : L \to \{ordinary, urgent, committed\}$ maps locations to location types. In *urgent* locations the outgoing transition will be executed immediately. *Committed* locations are useful for modeling of sequences of actions executed atomically without time passing.

The property specification language (aka model checking query language) is a subset of TCTL (Behrmann et al., 2004). For real-time applications, *time bounded reachability* is one of the fundamental properties. The state which satisfies formula φ and is reachable from model initial state is expressible using TCTL formula $\mathsf{A}\ \Diamond\varphi$ && $Clock \leqslant TB$, for the time bound TB. An extension of time bounded reachability is the reachability of a state relative to some preceding state in computation tree expressed using "*leads to*" operator denoted $ts \rightsquigarrow_{TB} rs$, for preceding state ts and reachable within time bound TB state rs.

3.3 Assume/Guarantee Contracts

The theory of contracts (Benveniste et al., 2018) has proven itself in series of works (Rouquette et al., 2023; Mercer et al., 2021). However, its use in MBT is just gaining more popularity (Aichernig et al., 2017; Badithela et al., 2023; Guin et al., 2022). Contracts specify components' interface properties representing the *assumptions* on their environment and the *guarantees* (regarding output) of the component under these assumptions. A/G contract being a pair (A,G) is said to be *consistent* if there is a component implementing it and *compatible* if there is an environment in which the contract can operate.

In this paper, we use three main *contract operators*. The *composition* of two interacting components is a partial function of contracts involving a *compatibility* criterion. Contracts C and C' are *compatible* if their shared variable types match and if there exists an environment in which they properly interact.

The second contract operator required when combining different view contracts of a component is the *conjunction* operator (denoted $\wedge$). The assume of contracts conjunction $C_1 \wedge C_2$ is the disjunction of both contracts assume part and the guarantee is the conjunction of their guarantees.

In this work, we instantiate the contracts metatheory by introducing a format for the contracts conjunction where conjuncts are extended with timing constraints defined as follows:

$$\bigwedge_{\rho\in\Omega} \left[S_p^{\rho} \wedge A_{\rho}\left(V^I\right) \rightsquigarrow_t S_q^{\rho} \wedge G_{\rho}\left(V^O\right)\right], \qquad (1)$$

where Ω denotes the set of contracts; S_p^{ρ} - (Control) pre-state of the Component; S_q^{ρ} - (Control) post-state of the component; A_{ρ} - Assumptions of ρ^{th} contract in pre-state S_p; G_{ρ} - Guarantees of ρ^{th} contract in post-state S_q; V^I denotes the vector of component's input variables and V^O that of output variables. The indexing of components is deferred unless it is clear from the context; $\rightsquigarrow_t$ denotes time bounded *leads to* operator in TCTL (note that *leads to* binding is weaker than that of conjunction).

Third contract operator concerns *refinement*. A contract C_1 refines a contract C_2, denoted by $C_1 \leq C_2$, if the assumptions of C_1 are relaxed $(A_2 \leq A_1)$ and its guarantees are strengthened $(G_1 \cup \neg A_1 \leq G_2 \cup \neg A_2)$.

4 FROM CONTRACTS TO MODELS

The test adapter generation process, as depicted in Figure 1, includes two steps. In Step 1, the Adapter is introduced as the new component in the test configuration model that divides the interface between components of Environment and SUT into two parts, Environment side and SUT side, so that adapter can be considered as a bidirectional repeater with delays due to its implementation. The Adapter in the model and its contract serve here as placeholders for further refinement with implementation. The preservation of integral correctness properties verified in earlier test development steps are assured by rerunning their satisfiability checks on the model supplemented with the Adapter to examine its caused delays. In Step 2, the abstract interfaces of the Adapter are refined to executable interfaces of SUT. The mappings between abstract and concrete interfaces are defined as transformer functions of the Adapter. The executable code is derived for each transformer so that the test can run with the adapter on a MBT execution platform (in this work on UPPAAL TRON). The transformers, the conjunction of which implements the Adapter model functionality, are derived as implementations of A/G contracts between Environment and SUT component described in detail in the following subsections.

4.1 Adapter Contract Extraction as Refinement of SUT-ENV Contract

The test adapter is a reactive component that is activated by input events. We group the Adapter behaviors by the sources of its input events. On the one hand, the SUT environment model M^{Env} generates symbolic inputs to SUT model. On the other hand, physical SUT emits concrete outputs as reactions prescribed as refinements of SUT model M^{SUT} symbolic outputs. Having such partitioning of Adapter inputs, the assume part of Adapter contract includes symbols of alphabet Σ_A^I, the union of two subsets $\Sigma_A^I = \Sigma_A^{I'} \cup \Sigma_A^{I''}$, the symbolic one $\Sigma_A^{I'}$, identical to Environment and SUT symbolic alphabets, i.e., $\Sigma_A^{I'} = \Sigma_{Env}^O = \Sigma_{SUT}^I$ and the concrete one $\Sigma_A^{I''} = \Sigma_{SUT}^{O''}$ of SUT output. The guarantee part, in turn, includes Adapter output symbols Σ_A^O which, like input alphabet, is the union $\Sigma_A^O = \Sigma_A^{O'} \cup \Sigma_A^{O''}$, where $\Sigma_A^{O''} = \Sigma_{SUT}^{I''}$ capturing SUT concrete input and $\Sigma_A^{O'}$ identical to M^{SUT} symbolic output Σ_{SUT}^O.

4.2 Adapter Model Generation

The adapter functionality is divided between sets of SUT input and output transformers (Figure 1). By the type of transformation map, they are: symbol to data transformers, with signature $\Sigma_A^{I'} \mapsto \Sigma_A^{O''}$; data to sym-

bol transformers, with signature $\Sigma_A^{I''} \mapsto \Sigma_A^{O'}$; symbol to behavior transformers with signature $\Sigma_A^{I'} \mapsto \Sigma_A^{O''*}$; and behavior to symbol transformers with signature $\Sigma_A^{I''*} \mapsto \Sigma_A^{O'}$. A transformer implements only unidirectional data/command flow. Symmetrically to SUT input transformers, the SUT output transformers implement M^{SUT} to M^{Env} data flow. They monitor SUT (physical) test output port and map the SUT concrete output values to symbolic ones in $\Sigma_A^{O'}$, i.e., implementing either the map $\Sigma_A^{I''} \mapsto \Sigma_A^{O'}$ or $\Sigma_A^{I''*} \mapsto \Sigma_A^{O'}$, where $\Sigma_A^{O'}$ symbols denote equivalence classes of SUT outputs.

The generation of models from contracts presumes a set of contracts, the instances of formula (1)), one for each transformer. In the course of parsing adapter contracts, the Algorithm 1 populates the Adapter component of the test model with model patterns where each represents a transformer composed together to a UTA process. Parsing of contracts presumes a machine-readable format

$Component{:}Name{:}[Assume]\ /\backslash\ Input\ /\backslash\ Pre-state --> Post-state\ /\backslash\ Output\ /\backslash\ [Guarantee]$, where the contract's *leads to* time bound is specified as a conjunct in the *Guarantee* part.

To process the contracts a parser is generated that checks the syntax and extracts the terms of the contract. Jflex and Byacc/J tools are used for lexical analysis and for parser generator, respectively. The grammar G_{TCTL} the parser accepts is that of TCTL expressions used in UPPAAL tool, with the difference that acceptance rules of certain terms are expanded with generative function to map the accepted nonterminal symbol to a corresponding UTA model element in XML format. The lexer identifies the specific parts of the contract as tokens that are used by the parser to verify the contract against its grammar.

The second part of the algorithm implemented in Java using UPPAAL libraries takes the parser output to generate the UTA model and TCTL queries for its correctness checking. The production rules corresponding to the upper part of the contracts parsing tree are depicted as follows.

⟨contract⟩ ⊨ `component:name:`⟨left⟩`-->`⟨right⟩
⟨left⟩ ⊨ ⟨assume⟩`/\input/\pre-state`
⟨right⟩ ⊨ `post-state/\output/\`⟨guarantee⟩
⟨assume⟩ ⊨ G_{TCTL}
⟨guarantee⟩ ⊨ G_{TCTL}

In the generative part of Algorithm 1 the parsed contract elements are mapped to UTA model shown in Figure 2 for the composition of two transformers.

The instantiated transformer models are composed into full Adapter model by merging their locations *Ready* and *Failure*. Location *Ready* denotes the waiting for the next input. The transition to auxiliary location *Failure* is not generated by the parser. It is default element to signal that its contract guarantee is not satisfied due to some adapter internal failure and not because of violating the test conformance relation.

Algorithm 1 generates the adapter model by parsing contracts one by one. It extracts the lhs of *leads to* and assigns its value to the *Guard* tag of transition ($Ready, Contract\_$.) in UPPAAL XML file. Source and target locations as well as other attributes of this transition are pre-defined by the transformer pattern and like location *Failure* and edges connected to it, they do not need instantiation by contract parsing. However, the invariant of location $Contract\_$. is parsed out from the contract where it specifies the upper time bound of transformers contract *leads to*. The outgoing from location $Contract\_$. edge is labeled in the guard condition with (optional) lower time bound of *leads to* operator. Other labels of this edge denote the refinement of the transformation function f^I or f^O in the *Assignment* tag and the update of the state variable S with next control state parsed out from the second conjunct in the contract's *leads to* rhs formula. The target location of this transition is anonymous *committed* location (denoted with c) introduced to separate the transformer function computation from the satisfiability check of contract guarantee instantiated with transformer function value. Therefore, one edge from *committed* to *Ready* location is labeled with guard meta symbol $Guarantee\_$. instantiated with rhs of the transformer's contract *leads to*. The other to *Failure* location has guard with the negation of the transformer's contract guarantee.

For clarity, we exemplify two same directional transformers' Tr_i and Tr_j composition in Figure 2. Generally, single port SUT input and output flows are processed by separate compositions of transformers, i.e., by the processes that model SUT input and output transformers, respectively. The same pattern scales to any number of SUT bidirectional test ports.

4.3 Adapter Model Correctness

In the provably correct test development workflow we show that 1) the adapter contracts do not violate the system level contracts and 2) that the adapter implementation satisfies its own local contract.

Thus, while each transformer function is specified

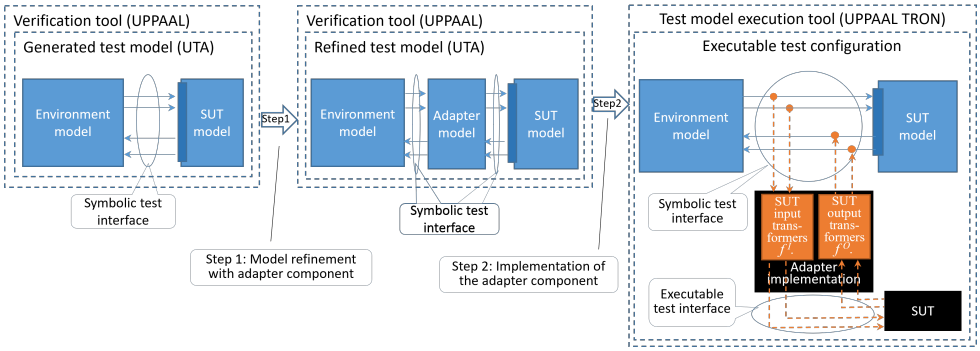

Figure 1: Adapter generation workflow.

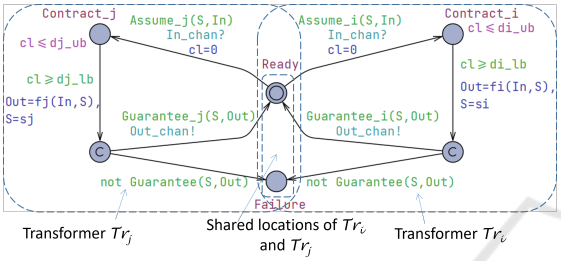

Figure 2: Transformers' templates composition at Step 1.

as a contract, the full contract of the adapter is a conjunction of transformers' contracts. Adapter model correctness with respect to its contract means satisfaction of following properties:

- P1 - *relative input completeness*. Adapter must accept all symbols in $\Sigma^{I'}_{SUT}$ and $\Sigma^{O''}_{SUT}$, i.e., $\Sigma^{I'}_{SUT} \cup \Sigma^{O''}_{SUT} \subseteq \bigcup_i dom f^I_i$.
- P2 - *responsiveness*. Adapter must *react* to all input symbols and timeouts with observable outputs, i.e., $\Sigma^{O'}_{SUT} \cup \Sigma^{I''}_{SUT} \subseteq \bigcup_i dom^{-1} f^O_i$.
- P3 - *reactive*. Adapter reaction time must be less than input interarrival times, i.e., adapter should not violate test model timing constraints by delaying and/or queuing the input streams.
- P4 - *uniqueness* of the mapping. Input symbols from different equivalence classes of input alphabet map to non intersecting co-domain symbol sets, i.e., $\forall i, j\ dom^{-1} f^O_i \cap dom^{-1} f^O_j \neq \varnothing \Rightarrow i = j$.
- P5 - *no unbounded waiting*. Reaction to missing responses of SUT must be mapped to timeouts, i.e., $f^O(\perp_{TO}) = TO$ where $\perp_{TO}$ denotes empty input during $TO \ll \infty$ time units and $TO \in \Sigma^{O'}_A$.

To assure the properties P1 to P5 cover also timeouts, the transformation function domains are extended with "time bounded silence symbol", $\perp_{TB}$, to denote the bounded waiting time. Properties P1 - P5 are formalized either as queries in TCTL or as auxiliary property acceptance automata composed with the test model and verified by the UPPAAL model checker.

4.4 Adapter Code Generation

Adapter code generation, as stated above, is a two step process. The first step is model generation which creates an UTA model and TCTL queries for checking correctness properties P1 to P5 from a set of contracts. The next step generates a canonical source code of the adapter in Java from the UTA model internal XML representation in which special tags are place holders for injecting executable code snippets that implement signaling, conditions and state transformations associated with the tag. The supplemented XML is used to generate the adapter source code. Further details of adapter code generation are elaborated in the following section.

5 CASE STUDY

The test adapter generation steps are exemplified on a case study of an industrial climate control system which has four components. Two of them implement user interfaces (UI) - Wall Mount Panel and Mobile application. The third unit, Controller, regulates the climate locally. The fourth component, Server, adds cloud access to mobile application of the system and implements supervisory control of controllers. Each Controller interacts directly with Server by a request-response format. The protocol specification covers functional, fault-tolerance and timing constraints. Each controller can initiate the session independently of others after a successful TCP connection is established with the Server. The protocol that defines the test scenario includes messages that are exchanged at controller - server connecting phase.

Transmission errors and delays may alter the exchange and the anomalous behaviors need to be covered by the test cases to show that Server disconnects

```
Function GenerateModel(Contracts cl):
    forall contract in cl do
        tl ← Parse(contract)
        cmp ← tl[Component]
        if cmp is not in Components then
            t ← AddTemplate(cmp)
            AddLocation(t,Ready)
            AddLocation(t,Failure)
            AddDeclaration(st)
            AddDeclaration(t, cl)
            AddProcess(t)
        end
        c ← tl[Name]
        l0 ← GetLocation(cmp, Ready)
        l1 ←
          AddLocation(cmp, concat(Contract_, c))
        l2 ←
          AddLocation(cmp, type=Committed)
        l3 ← GetLocation(cmp, Failure)
        t0 ← AddTransition(cmp, l0, l1)
        t1 ← AddTransition(cmp, l1, l2)
        t2 ← AddTransition(cmp, l2, l0)
        t3 ← AddTransition(cmp, l2, l3)
        forall symbol in tl[Assumption] do
            if symbol is not in Declarations
              then
                AddDeclaration(symbol)
            end
        end
        AddGuard(t0, tl[Assumption])
        AddGuard(t0, st==tl[Pre-State])
        AddSync(t0, tl[Input], in)
        AddUpdate(t0, cl=0)
        forall symbol in tl[Guarantee] do
            if symbol is not in Declarations
              then
                AddDeclaration(symbol)
            end
        end
        AddUpdate(t1, tl[Guarantee])
        AddUpdate(t1, st=tl[Post-State])
        AddSync(t2, tl[Output], out)
        /* add guarantees to
           transitions t2, t3          */
        AddOutGuards(t2, t3)
        AddQuery(tl)
    end
```

Algorithm 1: Model generation algorithm.

from the client Controller if it fails to receive *client hello* within first I_{TO} seconds after successful connection, Server disconnects if the *client hello* message data fields have incorrect values. Server shall wait for the first M_h bytes from Controller for I_{TO} seconds after the TCP connection is established and Controller must be able to send M_h bytes in parts within I_{TO} seconds. The connection procedure is completed if all aforementioned requirements are satisfied.

5.1 Adapter Contracts and Adapter Model Generation

The adapter generation is exemplified by two contracts T_2^c and T_2^s that specify a fragment of connection protocol the adapter is mediating. Contract T_2^c specifies Controller action of sending $i\_chello\_sent(id)$ message when being in *connecting* state and T_2^s specifies Server's reaction to it when receiving it.

$$T_2^c : o\_cnct(id) \wedge S_{cl}(id) = connecting \rightsquigarrow_{<I_{TO}} S_{cl}(id) = chello\_sent \wedge i\_chello\_sent(id)$$

$$T_2^s : i\_chello\_sent(id) \wedge C_h \leq I_{TO} \wedge data\_valid(id) \wedge S_{con}(id) = connected \rightsquigarrow_{<C_g} S_{con}(id) = shello\_sent \wedge o\_shello\_sent(id)$$

Contracts T_2^c and T_2^s are composed with Adapter contracts B_4^a and B_5^a, respectively, where

$$B_4^a : i\_chello\_sent(id) \wedge S_{ad}(id) = ready \rightsquigarrow S_{ad}(id) = processing \wedge tcp\_send(id, chello)$$

$$B_5^a : tcp\_receive(id, shello) \wedge S_{ad}(id) = processing \rightsquigarrow S_{ad}(id) = ready \wedge o\_shello\_sent(id).$$

Contract B_4^a states that when Adapter receives in state *ready* the input *i_chello_sent* from Controller then this leads Adapter to the state *processing*, emitting the concrete output *tcp_send(id, chello)*. The executable output action sends *client hello* message to Server. Contract B_5^a specifies the transformer that is triggered by receiving concrete output symbol *tcp_receive(id, shello)* sent from SUT. Then Adapter emits symbolic output *o_shello_sent*. The contracts to be implemented are:
adapterEnv:B4a: i_chello_sent /\ ready --> processing /\ tcp_send_chello /\ [cl <= 3 /\ cl >= 2]; and
adapterSut:B5a: tcp_receive_shello /\ processing --> ready /\ o_shello_sent /\ [cl <= 3 & cl >= 2];

The clock *cl* is added by Algorithm 1 to each component so that it is reset on arrival of inputs. In the above example contract the [*Assume*] field is not shown because the assumption in this contract contains only symbol *i_chello_sent* in the input.

The generated SUT input and output Adapter implementation models for contracts B_4^a and B_5^a are shown in the left and right part of Figure 3, respectively. The parser extracts the component name

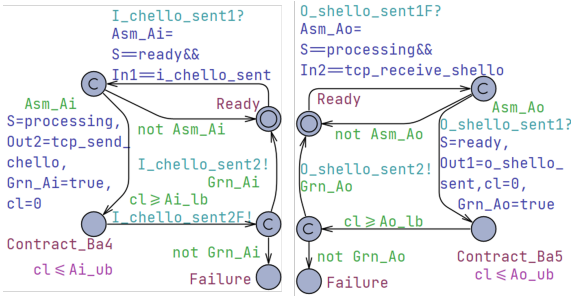

Figure 3: Transformer models of contracts B_4^a and B_5^a.

adapter and assume/guarantee terms from its contract. The algorithm generates the transformer pattern by populating it with extracted conditions and terms it includes. Variables corresponding to the input and output symbols, e.g., *i_chello_sent* and *tcp_receive_shello* are declared by Algorithm 1 to generate the UPPAAL TCTL queries that correspond to contracts above as follows:

B_4^a: *i_chello_sent_b and adapterEnv.st == Ready −− > adapterEnv.st == Processing and tcp_chello_sent_b and* $(cl <= 3$ *and* $cl >= 2)$

B_5^a: *tcp_receive_shello_b and adapterSut.st == Ready −− > adapterSut.st == Ready and o_shello_sent_b and* $(cl <= 3$ *and* $cl >= 2)$

The queries B_4^a and B_5^a generated by Algorithm 1 involve auxiliary variables, e.g., *i_chello_sent_b* which are set to true by the model when the corresponding channel, e.g., *i_chello_sent* triggers the transition labeled with corresponding input action. The queries generated are used for checking the correctness of the adapter implementation model.

5.2 Adapter Code Generation

Algorithm 1 generates the model that is input for adapter code generation by Algorithm 2. The source code in Java enables its integration with test execution environment DTRON, the extension of UPPAAL TRON tool. The key element created in the process is the symbol map that provides symbols associated with the abstract input from the environment model. They are i) concrete input to the SUT, ii) Concrete output from SUT and iii) abstract output to the environment model. The *AddHandler* method generates the handler that is triggered for a specific DTRON input during the online test execution.

```
private boolean tcp_send_chello(
 IDtronChannelValued v) {
  //Send concrete input to SUT
  return true;}
public void addListeners()
 throws SpreadException {
  IDtronChannel ch_chello_sent =
new DtronChannel("chello_sent");
 getMBTDtron().addDtronListener(
  new DtronListenerExt(ch_chello_sent) {
   @Override public void messageReceived(
    IDtronChannelValued v) {
     if(tcp_send_chello(v)) {
       IDtronChannel reply =
        new DtronChannel("shello_sent");
       IDtronChannelValued valued =
        reply.constructValued(
        (Map<String, Integer>) null);
       send(valued);
  }}}});}
```

The adapter code snippet in the above listing is generated for contracts B_4^a and B_5^a. The abstract input symbol *i_chello_sent* is registered as a channel for monitoring. The handler in the method *addListeners* is triggered when *i_chello_sent* is received. The action of sending the concrete input to the SUT is performed in the method *tcp_send_chello* named after the input symbol sent from controller to server over TCP. The adapter sends the abstract output symbol *o_shello_sent* if the method *tcp_send_chello* succeeds.

Function `GenerateAdapter(`*Contracts cl, Component ready*`)`**:**
- *um* ← `GenerateModel(`*cl*`)`
- *sm* ← `GetSymbolMaps(`*cl*`)`
- *tr* ← `GetTemplate(`*um, ready*`)`
- **forall** *transition* ***in*** *tr* **do**
  - *L* ← `FromLocation(`*transition*`)`
  - *S* ← `GuardState(`*transition*`)`
  - **if** *L* ***is*** *Ready* ***and*** *S* ***is*** *ready* **then**
    - *inp* ← `Input(`*transition*`)`
    - *cIn* ← `GetConcreteInput(`*sm, inp*`)`
    - *cOut* ← `GetConcreteOutput(`*sm, inp*`)`
    - *out* ← `GetAbstractOutput(`*sm, inp*`)`
    - `AddHandler(`*inp, cIn, cOut, out*`)`
  - **end**
- **end**

Algorithm 2: Adapter generation algorithm.

6 CONCLUSION AND FUTURE WORK

The MBT method proposed in this paper allows automatic generation of test adapters from SUT interface Assume/Guarantee contracts reducing so the manual coding effort and avoiding human errors in that. However, deriving the adapter contracts needs some expert's assistance. While symbolic test I/O alphabets

used in adapter contracts can be extracted automatically from the SUT model, their mapping to executable inputs and observable outputs is dependent on abstraction function used in SUT modeling. As a partial solution we generate instantiated adapter contract templates with place holders from the SUT model where the executable test data are therafter manually filled in. Therefore, our future target is applying ML for automatic learning of the mapping between abstract and concrete I/O alphabets to fill this gap.

REFERENCES

Aichernig, B., Hörmaier, K., Lorber, F., Nickovic, D., and Tiran, S. (2017). Require, test, and trace IT. *International Journal on Software Tools for Technology Transfer*, 19:409–426.

Alur, R., Courcoubetis, C., and Dill, D. (1990). Model-checking for real-time systems. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425.

Badithela, A., Graebener, J. B., Incer, I., and Murray, R. M. (2023). Reasoning over test specifications using assume-guarantee contracts. In Rozier, K. Y. and Chaudhuri, S., editors, *NASA Formal Methods*, pages 278–294, Cham. Springer Nature Switzerland.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

Behrmann, G., David, A., and Larsen, K. G. (2004). A Tutorial on UPPAAL. In Bernardo, M. and Corradini, F., editors, *Proceedings of Formal Methods for the Design of Real-Time Systems, SFM-RT*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer.

Bensalem, S., Bozga, M., Nguyen, T.-H., and Sifakis, J. (2010). Compositional verification for component-based systems and application. *IET Software*, 4:181–193(12).

Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T. A., and Larsen, K. G. (2018). Contracts for system design. *Found. Trends Electron. Des. Autom.*, 12(2–3):124–400.

Blundell, C., Giannakopoulou, D., and Pă săreanu, C. S. (2005). Assume-guarantee testing. In *Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems*, SAVCBS '05, page 1–es, New York, USA. Association for Computing Machinery.

Boudhiba, I., Gaston, C., Le Gall, P., and Prevosto, V. (2015). Model-based testing from input output symbolic transition systems enriched by program calls and contracts. In El-Fakih, K., Barlas, G., and Yevtushenko, N., editors, *The 27th IFIP International Conference on Testing Software and Systems (ICTSS-2015)*, pages 35–51. Springer.

Ciupa, I. and Leitner, A. (2005). Automatic testing based on Design by Contract.

Guin, J., Vain, J., Tsiopoulos, L., and Valdek, G. (2022). Temporal multi-view contracts help developing efficient test models. *Baltic Journal of Modern Computing*, 10(4):710–737.

Güldali, B. (2014). *Integrating Contract-based Testing into Model-driven Software Development*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn.

Kumar, A. (2023). Formalization of structural test cases coverage criteria for quantum software testing. *International Journal of Theoretical Physics*, 62.

Larsen, K. G., Mikucionis, M., and Nielsen, B. (2005). Online testing of real-time systems using UPPAAL. In Grabowski, J. and Nielsen, B., editors, *Formal Approaches to Software Testing, FATES*, volume 3395 of *Lecture Notes in Computer Science*. Springer.

Mercer, E., Slind, K., Amundson, I., Cofer, D., Babar, J., and Hardin, D. (2021). Synthesizing verified components for cyber assured systems engineering. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 205–215.

Meyer, B. (1988). Eiffel*: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246.

Milicevic, A., Misailovic, S., Marinov, D., and Khurshid, S. (2007). Korat: A tool for generating structurally complex test inputs. In *29th International Conference on Software Engineering (ICSE'07)*, pages 771–774.

Rouquette, N., Incer, I., and Pinto, A. (2023). Early design exploration of space system scenarios using assume-guarantee contracts. In *2023 IEEE 9th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 15–24.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312.

Vain, J., Kanter, G., and Srinivasan, S. (2017). Model based testing of distributed time critical systems. In *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pages 99–105.

Wijesekera, D., Ammann, P., Sun, L., and Fraser, G. (2007). Relating counterexamples to test cases in CTL model checking specifications. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, A-MOST '07, page 75–84, NY, USA. ACM.

Xu, D., Xu, W., Tu, M., Shen, N., Chu, W., and Chang, C.-H. (2016). Automated integration testing using logical contracts. *IEEE Transactions on Reliability*, 65(3):1205–1222.