An Intuitive Approach for Specifying Interface Constraint
Qianxiang Wang

School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies (Peking University)/Ministry of Education
Beijing, China, 100871
wqx@pku.edu.cn

Abstract

Interface is important for software modules, e.g., classes, components and services. Most current description approaches focus only on each method signatures of the target interface, while constraint on interface, e.g. temporal sequence of the invocation is less considered so far. This paper proposes an intuitive approach for specifying interface constraint, especially the temporal constraints on different methods of the interface. As a directed graph, the proposed approach is intuitive and powerful. Notations of relationships are introduced, followed by some examples. One algorithm that can translate the high level intuitive graph to low level FSA is presented.

1. Introduction

Interface is important for software modules, e.g., classes, components and services. Interface describes how software module can be used by the caller program. It can help programmers to enhance the safety of both caller and callee program. The key part of interface description is the methods signatures of the target module. For distributed system, several interface description languages have been proposed by different organizations, e.g. IDL by OMG [13], MS-IDL by Microsoft [14], Java interface by SUN [17], WSDL by W3C [18].

Signature defines the message format between interacting modules. Most current interface describing approaches just list all method signatures one by one [13, 14, 17, 18]. Even some security and transaction related information is processed as one kind of special parameters of method. But contract between the caller program and the callee program is beyond method signature. One of them is the temporal constraint on different methods. That means, by which order should the methods be called. For example, J2SDK1.3.1 contains 914 classes, out of which 81 classes were found had method sequences constraint [20]. If the invocation order is violated, then the callee may not run correctly, and the caller can’t get right result.

To make it clear, let’s consider one example ShoppingCart, which is a widely used class in web-based shopping system. Figure 1 lists the interface of ShoppingCart. For simplicity, we list only methods that are related with temporal constraints: “addItem()”, “setClientInfo()” and “checkout()”.

```java
interface public class ShoppingCart {
    public Integer addItem(Integer n, String p);
    public void setClientInfo(String info);
    public Double checkout();
    ……
}
```

Figure 1. The interface of ShoppingCart

The temporal constraints on this interface include: 1) before invoking method “checkout()”, the caller must have invoked method “addItem()” (added some product in the shopping cart), and have invoked method “setClientInfo()” (filled in the necessary information such as credit card number and mail address). 2) when the method “checkout()” has just been invoked, it cannot be invoked again, until one of method “addItem()” and “setClientInfo()” is invoked.

1.1 Current Approaches

To cope with temporal constraints on interface, Strom and Yellin proposed typestate to extend the notion of type, which arises from the realization that, at any point in time, the operations that can be performed on a variable depend not only on the type of the variable, but also upon the state of the variable [24]. They used Nil and Hermes language to describing the temporal constraints. Ramalingam et al. proposed a more general language Easl to specifying an abstract semantics for a component library [25].

LTL (Linear Temporal Logic) is widely used to express the temporal properties. It has been used also to express interface temporal constraint [3]. Another similar approach is ERE [9]. LTL is powerful and precise in express temporal constraint. But the main shortcoming of this traditional approach is that they are not intuitive. It is difficult for users to express requirement by LTL [23]. From another view, LTL is suitable to express constraints on properties, not for event like method invocation. Property usually holds during some period of time, while event occurs at some point of time. Multiple properties can hold together, but multiple events can’t occur at the same time. TL Chart [5] is a visual specification language that combines the visual and intuitive appeal of non-
deterministic Harel Statecharts with formal specifications written in Linear-time Temporal Logic, but this extension is not suit for interface.

If we consider a method invocation as an event, then constraint on method invocation order can be considered as constraint on event sequence. Event Graph was introduced by Schruben [15] to develop simulations of discrete-event systems. The system dynamics are characterized by events that change the state of the system and the logical and relationships among these events. Event Graph emphasizes condition of event occurrence. Belli proposed Event Sequence Graph (ESG) to model GUI, and imported “next is not” relationship in graph [1]. Still, it expresses only “future” relationship; “past” relationship is not included. Meanwhile, there are too many edges in the graph under some special situations.

The approach proposed in this paper is a high level specification, which is a graphical approach and can express most temporal interface constraint. It is easy to specify and easy to understand. The proposed approach is powerful. It can cover most of the common temporal constraints [23]. As a formal high level constraints specification, it can be translated into low level expressions, e.g., Finite State Automata (FSA), which is widely used for program verification.

1.2 Applications of Interface Constraint

The basic application of the proposed approach is in designing time: 1) Call ee developer can provide more information about contract between caller and callee; 2) The caller developer can understand the callee better with detailed interface information.

The more important application of formal temporal constraint is to automatically find temporal constraint violations on both caller program and callee program.

(1) Verifying caller program in static or dynamic ways. By analyzing program statically, it is feasible to find whether the requests are sent following the temporal constraint [26,27]. Meanwhile, most runtime verification approaches [2,3] rely on specification of safety property.

(2) Generating more efficient test cases for callee program, like work of Belli [1] and Korel [22]. Those test cases can be used to verify whether the program runs consistently with the predefined constraint. For example, can program find wrong invocation sequence? Can program accept all right invocation sequence?

(3) Monitoring program online. Even after the callee program has been deployed in the target environment, the specification can still be used to verify incoming request messages [19]. It is very important for safety and security of network based software. Nowadays, most web-based server use JSP pages that will be shown to users. This structure guaranteed the invocation order by predefined page structure. But in other invoking scenarios such as Web Services, there is no mechanisms to guarantee the invocations order from client.

The rest of this paper is organized as follows: Section 2 introduces foundation of the proposed approach, especially notations for the graph. Section 3 analysis some features of the proposed approach. Section 4 describes an algorithm of translating ECG to FSA, and the monitoring code. Section 5 introduces related work. Section 6 give some discusses and concluded the paper.

2. Foundation of the Approach

The key idea of the proposed approach is that the constraint on interface is combined by constraints on every method of the interface. So one basic assumption in this paper is: “The temporal constraint on one interface can be express as the aggregation (‘AND’ relationship) of constraint on every method”.

When we focus on constraints on each method, and want to express the temporal constraints on this method, it is very natural that the focused method should sit on the center of the diagram. This lead to a novel graph: the method is taken as vertexes of the graph, while temporal relationships with other methods are taken as edges. This is different with FSA: the state is taken as vertexes, while methods are taken as edges between states. To make the graph more general, we call such graph Event Constraint Graph (ECG for short).

The proposed approach imports four notations to express temporal constraints: 1) “the next event must be some event”; 2) “some event must happen in the future”; 3) “the last event must be some event”; 4) “some event must have happened in the past”. Beside these four temporal relationships, the propositional relationships (“AND”, “OR” and “NOT”) between these four temporal relationships are expressed also.

2.1 Basic Symbols for Temporal Constraints

This section introduces symbols that are used to express the four focused temporal constraints.

2.1.1 Constraint on Future

We use two notations “→” and “→→” to express future relationships “next event must be some event” and “some event must happen in the future”. Figure 2(a) means “If e has just occurred, then the next event must be e1”. Figure 2(b) means “If e has just occurred, then e1 must happen in the future before the end of the sequence”.

![Figure 2. Notations for constraint on future.](image-url)
2.1.2 Constraint on Past

We use two notations “\(\rightarrow\)" and “\(\rightarrow>\)" to express past relationships “last event must be some event” and “some event must have happened in the past”. Figure 3(a) means “If ‘e’ has just occurred, then the last event must be ‘e1’”. Figure 3(b) means “If ‘e’ has just occurred, then event ‘e1’ must have happened in the past”.

![Figure 3. Notations for constraint on past.](image)

2.2 Propositional Options

This section introduces symbols that are used to express the propositional options “AND”, “OR” and “NOT”.

2.2.1 OR

For some specific event, the default expression (no special notation) for propositional relationships on different future constrains is “OR”, and the default past propositional relationships is “OR” too. Figure 4(a) means: “if ‘e’ has just occurred, then ‘e1’ must happen as the next event OR ‘e2’ must happen later before the end of sequence”. Figure 4(b) means: “if ‘e’ has just occurred, then ‘e1’ must have happened as the last event OR ‘e2’ must have happened”.

![Figure 4. Notations for “OR” relationship.](image)

2.2.2 AND

For every specific event, the “AND” propositional relationships for constraints on future or relationships for constraint on past is marked explicitly by the joint notation “\(\land\)”. This notation connects several constraints on the same interface into one constraint group. Figure 5 (a) means: if ‘e’ has just occurred, then ‘e1’ must happen as the next event AND ‘e2’ must happen later before the end of sequence”. We can also express this constraint as: C = AND (FN (e, e1), F (e, e2)). Figure 5 (b) means: “if ‘e’ has just occurred, then ‘e1’ must have happened as the last event AND ‘e2’ must have happened”.

![Figure 5. Notations for “AND” relationship.](image)

Beside this explicitly marked “AND” relationship, there are two other “AND” relationships that are hidden in ECG. One is the “AND” propositional relationships for every constraint on different event, as the assumption mentioned in section 1.3. The other is the propositional relationships for future constraint sets and past constraints set on specific event, see Figure 6. It belongs to “AND” relationship also.

![Figure 6. Default “And” relationship for constraints on future and constrains on past.](image)

2.2.3 NOT

We import notation “\(\neg\)” to express “NOT”, so as make the model easier to understand. Figure 7(a) means “If ‘e’ has just occurred, then the next event must NOT be ‘e1’”. Figure 7(b) means “If ‘e’ has just occurred, then ‘e1’ must NOT happen in the future before the end of the sequence”.

![Figure 7. Notations for “NOT”](image)

2.3 START and END

To express constraints related with the event sequence head and event sequence tail, we import following two special symbols to express two special “event”: “no event” before event sequence and “no event” after event sequence. No event before event sequence (“START” in short): ●. No event after event sequence (“END” in short): ☐.

2.4 Two examples

With notations introduced above, we now give two examples to show how ECG can express interface constraints intuitively. The first one is the example introduced in Section 1. Figure 8 shows the corresponding ECG.

![Figure 8. ECG for example in section 1.](image)

The second example is from [20]. The constraint is about java.security.Signature: If sign() is called before initSign() is called, an exception is thrown. Likewise, if verify() is called before init Verify() is called, an exception is thrown also. If update() is called before initSign() or
3. Feature Analysis for ECG

After having seen some examples of ECG, now let’s turn to some common features of ECG, to get a better understanding of ECG.

3.1 Confliction Detection

As one formal expression, ECG can be used to find some conflictions. Conflictions may be caused by different reasons. “AND” and “cycle” are the most popular cases that lead to conflictions.

3.1.1 “AND” leaded Conflictions

Constraint expressed by Figure 10(a) means: “When ‘e’ has just occurred, then the next event must be ‘e1’” and “When ‘e’ has just occurred, then the next event must be ‘e2’”. Because only one event can occur in event sequence, so this graph leads to an error expression: which event must happen next? Similar errors are shown in Figure 10(b), Figure 10(c), Figure 10(d).

3.1.2 Cycle leaded Conflictions

In Figure 11, all cycles will have conflictions if they have no outlet or inlet edges. Constraints that are expressed by Figure 11(a, c, e) will lead to one event sequence that can not end. Constraints that are expressed by Figure 11(b, d, f) will lead to one event sequence that can not find start event. There are two solutions for cycle confliction: one is add one outlet (inlet) edge, the other is erase one edge to broken the cycle.

4. Translating ECG to FSA

As we mentioned in section 1, one important application of ECG is to verify program, especially the method invocation sequence: whether the invocation sequence is valid. Although high level constraints specification such as ECG is easy to understand, it is not convenient for verification: if there are many constraints, it is difficult to verify whether the incoming invocation conflict with some constraint, because we have to check this invocation with each related constraint.

If we can generate one FSA that uses global state as vertex, and the vertex set equals to all possible state set, then we can verify incoming invocation easily. FSA use all legal input events as edge. For every state, all legal incoming events and the target states are listed explicitly, under the high level constraint.

The principle of generating FSA from ECG is to list all potential states by combining state of different event, then reduce unnecessary states, and then find all transitions between states.
4.1 Translating Algorithm

```c
1  int initial_state_num = get_FN_PL(constraint,se);
2  int n = get_FF_PP(constraint,se);
3  int m = get_sensitive_events(constraint);
4  generateFSA(constraint) {  
5      int m = get_sensitive_events(constraint);  
6      int n = get_FF_PP(constraint,se);  
7      for (int i = 0; i < states_num; i++)
8          calculate_next_state();
9          check_constraint();
10  }  
11  for (int j = 0; j < events_number; j++)
12      for (int i = 0; i < states_num; i++)
13          calculate_next_state();
14  delete_error_states();
15  label_PP();
16  label_end();
17  label_direct();
18  label_begin();
19  label_direct();
20  fill_coded_states();
21  int coded_state = new int[initial_state_num][column_num];
22  for (int i = 0; i < states_num; i++)
23      for (int j = 0; j < events_num; j++)
24          coded_state[i][j] = new int[coded_state[i][j]];  
25  fill_coded_states();
26}
```

This is the pseudo code for the translating algorithm.

The algorithm has been implemented, and can be found via: http://www.sei.pku.edu.cn/~wqx/mass/ECG2FSA. In this Figure 13, we use pseudo code to show the translating algorithm. The translating algorithm includes four main steps: construct initial state table, delete conflicted states, construct translated table, and reduce table.

4.1.1 Construct initial state table

See line 1-4. The “sensitive events” in the algorithm means those events that affect some other events in the sequence, “e” in Figure 2, “e1” in Figure 3, and “e” in Figure 4(a) are all sensitive events. The completely combination (without considering the order) of these event forms the initial state set. Number “m” in the algorithm means the number of “happened” events. These events can form 2\(^m\) states. Number “n” in the algorithm means the number of “just occurred” events. The state number that formed by those “m + n” events is “1 + 2\(^m\) * (n+1)”, in which “1” means nothing happened. Each state represents one case that “which event has just occurred, and what events have happened before”.

4.1.2 Delete conflicted states

See line 5-7. Many of the “1 + 2\(^m\) * (n+1)” states formed in step one are invalid states. Before generating FSA, we must delete them. There are 4 main kinds of invalid state: 1) direct conflictions. That means, in this state, the just occurred event was label “not happened”. See state “1, 2, 5, 7” in Figure 14(a); 2) “START” related conflictions. That means, sequence can only started with some method, but in some states, some other event happened; 3) “END” related conflictions. That means, in some state, some event happened after event that must be the last event; 4) indirect conflictions. That means, current event occurs, but precondition event(s) did not happened. See state “9, 10, 11” in Figure 14(a).

4.1.3 Construct initial translated table

See line 8-14. This is the main part of algorithm. We construct initial FSA first. Then process each state: according to the current state (“which event has just occurred, and what events have happened before”). For each event, we check whether it is valid to occur as next event, following each constraint. If it is valid, then we calculate: when the current event occurred, which is the next state that the FSA will enter.

4.1.4 Reduce table

See line 15-22. As the last step, we check the initial translated table and reduce the table. There are two main works. The first is to delete states that can not arrive in from the first state (no event happened). The second is to merge states that have same transitions with others. For two states that have “same transitions”, we mean in these two states, for each incoming event, they will enter the same new states. Merge these state can reduce FSA, but does not affect the graph semantics. After having reduced some states, we get one final FSA that is equal to the high level ECG in semantics.

4.2 Example to show the translating process

In this section, we explain the algorithm with the ShoppingCart example that is shown in Figure 1. We list three tables that were created during the generating process. Those three table corresponds with the results of step 1, 3, and 4. In these table, we use “a” to denote the method “addItem()”, “c” to denote the method “setClientInfo()”, and “e” to denote the method “checkout()”.

For Shoppingcart example, there are two events that are sensitive for “happened” constraints: “a” and “s” (so m = 2). And there are two events that are sensitive for “Just occurred” constraints: “a” and “s” (so n = 2). Then the total number of possible states for this example is 1 + 2\(^2\) * (2+1) = 13, see Figure 14(a). These 13 states list all possible combinations of the sensitive events in all constraints.

In Figure 14(a), state 0 means “no event happened”, state 1 means “‘a’ has not happened, ‘s’ has not happened, and ‘a’ has just occurred”. So this state is an invalid state because of event “a”: it has not happened, and it has just occurred! By similar analysis, we can find that state 2, 5, 7 are all invalid states. For state 9, “’s’ has not happened, ‘s’ has not happened, and ‘c’ has just occurred”, this situation is conflict with constraint: “if ‘c’ occurred, ‘a’ and ‘c’ must have both happened”. So state 9 is an invalid event. By
By deleting invalid states “1, 2, 5, 7, 9, 10, 11” from Figure 14(a), 6 states are left, see left half of Figure 14(b). For these 6 states, we calculate the next state for each incoming event. For example, in state 0, when “a” occurred, it will enter state 1: “‘a’ has happened, ‘s’ has not happened, and ‘a’ just occurred”. Thus we fill “1” in raw 3, column 5. In this table, “-1” means invalid invocation.

When verifying Figure 14 (b), we can find that state 2 and state 4 have the same transition set: when event “a”, “s”, and “c” come separately, both of them will enter state “2”, “4”, and “5” separately. So we can merge these two states, and get the final FSA table, see Figure 14(c).

Figure 15(a) expressed the generated FSA graphically. For each state, all incoming event and target state are listed explicitly. If we list only legal transitions (ignore illegal transitions), then we can get one simplified graphical FSA, see Figure 15(b).

Figure 15. FSA generated from model shown in Figure 8.

5. Related Work

Our work on the graphical specification of interface constraints was mainly motivated by work of Whaley et al. [20] and Belli [1], while work of translating ECG to FSA is mainly motivated by work of rewriting algorithms, which is implemented in Maude [8]. Both Whaley and Belli...
proposed graphical specification of interface constraints. And both of the proposed approach used method (not state of the instance) as vertex of the graph. The main difference between the approach proposed in this paper and the former proposed approaches is the four kinds of edges, which are used to express four relationships of methods. Our approach can be thought as the extension to those approaches, and thus can express more constraints.

Following Ernst’s work [6] which tries to discover invariants such as relations between variables, Whaley et al. try to find method call sequences for specified interface. They proposed a finite state machine for each field of a class, with one state for each method that writes that field [20]. They add restrictions on from which states methods that read the field may be invoked. They proposed and implemented two techniques that automatically extract such models. The first is a dynamic instrumentation technique that records legal method sequences from working programs. The second is a static analysis that infers pairs of methods that cannot be called consecutively. They have also developed a dynamic model enforcer to ensure that a given model is obeyed. Based on “next is” relationship, Whaley import dotted lines to express affected event. But this new notation is not an accurate description, so it is difficult to generate monitoring code from it. We have interesting with how to express constraints intuitively and accurately, and how to use them efficiently.

L. White introduced an FSA-based method for GUI testing, including a convincing empirical study to validate his approach [21]. Based on the proposed Event Sequence Graph, Belli extend L. White’s approach by taking not only desired behavior of the software into account, but also undesired situations [1]. ESG is used to test GUIs not only through exercising them by means of test cases which show that GUI is working properly under regular circumstances, but exercising also all potentially illegal events to verify that the GUI behaves satisfactory also in exceptional situations.

Runtime verification is gaining many researchers’ attention in recent years. In [8], Rosu presented a study in using rewriting in runtime verification and monitoring of systems. using rewriting to generate automata-like monitors, called binary transition tree finite state machines (abbreviated as BTT-FSAs), so as to effectively and efficiently evaluating LTL formulae on finite execution traces online, that is, by processing each event as it arrives. In their approach, initial constraints are expressed by LTL, and generating algorithm is an increase based. Meanwhile, Past Time LTL (PTTTL) and Future Time LTL (FTTTL) are seldom used together. As we mentioned in section 1, the main shortcoming for LTL is that, LTL is too abstract, not intuitive. So it is difficult for programmers to learn LTL, use LTL to specify interface constraint, and understand constraints that are specified with LTL.

The Java Modeling Language (JML) is a behavioral interface specification language that was used to specify the behavior constraint of Java modules [11]. It combines the “design by contract” approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus. In JML, specifications are written in special annotation comments, which are embedded in java class source code. Compared with JML, the graphical specification proposed in this paper is a light weight, intuitive approach.

For other general constraints on interfaces, such as pre-conditions or post-conditions, OCL [12] provides a good approach to express them. We hope our approach can be combined with OCL in some suitable way, so as to express constraint more precisely and intuitively.

6. Discussion and Conclusion
An intuitive approach for constraint specification is very important for application of constraint. If the constraints are difficult to specify, it is very hard to ask programmers to specify them formally. Actually, that is the reason why UML become prevalent in recent years.

Although ECG can express most of the common constraint [23], it should be extended to multiple directions, so as to express more constraints. For all potential extensions, we have strong interesting to those that express more information and can be used to generate code automatically. Here we list two of them that we may consider in the future.

(1) Timed Constraint
Many graphical specifications such as Event Graph [15] include detailed time to provide more information. Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real time systems [2]. MTL extends LTL by supporting the specification of relative-time and real-time constraints. With MTL, all four LTL future-time operators can be characterized by relative-time and real-time constraints specifying the duration of the temporal operator. ECG can be extended by real time information also.

(2) Scoped Constraint
Our current approach is event pair based: both of condition and result in single constraint is single event, although some constraint involves multiple events by combining multiple constraints. Dwyer proposed concept scope to express more complex constraints [23]. Similar work can be found in [10], which involves over 400 coding patterns for J2EE application. Yahav proposed an approach to verifying typestate that involves multiple objects [16]. To express more constraint, ECG should be extended in this direction.

In general, this paper proposed a graphical specification of temporal interface constraint and related
algorithm of translating the specification to FSA. The proposed graph is intuitive and powerful. It uses methods of interface as vertexes, and import four kinds of edges to represent four kinds of temporal relationship of event, two for future time and two for past time. Propositional relationship “AND” “OR” and “NOT” are considered also. Notations of these representations are introduced, followed by some features of the graph, such as cycle, confliction, redundancy, etc. One implemented algorithm that can translate the graph to FSA is introduced. We illustrated the graph and relation FSA by some examples. Those examples show that our approach is feasible.

Acknowledgement

This work was supported in parts by grants from the National High-Tech Research and Development Plan of China (No. 2006AA01Z175), the National Grand Fundamental Research Program of China (No. 2005CB321805), National Natural Science Foundation of China, (No. 60773160).

The author would like to thank Professor Aditya P. Mathur at Purdue University for hosting the author as a visiting scholar when this research work was explored, and for providing so much help in the preparation of this paper.

References