Contractor.NET: Inferring Typestate Properties to Enrich Code Contracts

Edgardo Zoppi, Víctor Braberman, Guido de Caso, Diego Garbervetsky, Sebastián Uchitel
Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
{ezoppi, vbraber, gdecaso, diegog, suchitel}@dc.uba.ar

ABSTRACT

In this work we present CONTRACTOR.NET, a Visual Studio extension that supports the construction of contract specifications with typestate information which can be used for verification of client code. CONTRACTOR.NET uses and extends CODE CONTRACTS to provide stronger contract specifications. It features a two step process. First, a class source code is analyzed to extract a finite state behavior model (in the form of a typestate) that is amenable to human-in-the-loop validation and refinement. The second step is to augment the original contract specification for the input class with the inferred typestate information, therefore enabling the verification of client code. The inferred typestates are *enabledness preserving*: a level of abstraction that has been successfully used to validate software artifacts, assisting in the detection of a number of concerns in various case studies including specifications of Microsoft Server protocols.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*validation*

General Terms

Algorithms, verification

Keywords

Typestate inference, contract strengthening, enabledness abstractions

1. INTRODUCTION

Design by contract is a programming discipline that prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. CODE CONTRACTS [5] is a Microsoft Research project that brings the advantages of design-by-contract programming to all .NET based programming languages, enabling the use of contracts without requiring a specific compiler. These contracts act as documentation which can be used to improve the quality of software via run-time checking and static verification of client code conformance to contracts.

The CODE CONTRACTS project focuses on contracts specifying requires clauses over methods parameters rather than requires clauses (and ensures clauses for that matter) over the object state: Few classes are provided with specifications that describe the state in which an object must be in when a method is executed. This is not surprising, contracts that refer to a shared data structure used by various methods are not as easy to write nor validate. Contracts combine in unexpected ways when sequences of methods are invoked, leading to inaccurate documentation and unintended specified behavior, this in turn leads to problems when trying to assure the quality of client code.

Typestate specifications [4] prescribe the possible coarse grained states in which an object can be during its lifetime. They are used to enforce safety properties that depend on changing object states. Typestates can be used to check conformance of client code for a class (i.e., enforcing that clients always perform valid invocation sequences over an API). This is typically assured using type-checking techniques (e.g., the FUGUE protocol checker [4]) or by encoding the typestates as state machines to be checked with the aid of a software model checker (e.g., the SLAM SDV [1]).

Typestate specifications, though applied in particular settings such as verification of lower level API client code, are not widespread. We believe that typestate specifications can support documentation of class behavior and verification of client code in a wider context and that the vision pursued by approaches such as CODE CONTRACTS can be strengthened by the use of typestates.

CONTRACTOR.NET[1] is a Visual Studio extension that supports the construction of contract specifications with typestate information which can be used for verification of client code. Internally, CONTRACTOR.NET features a two step process. First, source code is analyzed to extract a likely-typestate that is amenable to human-in-the-loop validation and refinement. The second step is to decompose on a per-method basis the final typestate back into a contract specification which can be used in the context of the CODE CONTRACTS project for verifying client code. The typestate itself can be used as documentation to support client code developers understand the contract specification.

[1]Available from: `http://lafhis.dc.uba.ar/contractor.net`

TOPI '11, May 28, 2011, Waikiki, Honolulu, HI, USA

The produced abstractions are *enabledness preserving* [3]: Behavior models which group class instances according to the methods they enable. Such abstractions have been successfully used to validate software artifacts both in the form of contracts and source code, assisting in the detection of a number of concerns in various case studies including specifications of Microsoft Windows Server protocols.

The rest of this paper is structured as follows: §2 presents an illustrative example of the CONTRACTOR.NET tool usage. In §3 and §4 we present the details of the two steps that comprise our approach: typestate inference and contract strengthening, respectively. §5 gives a brief description of the CONTRACTOR.NET Visual Studio Extension. The article concludes with a brief related work overview in §6 and final remarks in §7.

2. MOTIVATION

We now discuss a trivial example to illustrate how CONTRACTOR.NET can aid developers. Consider the scenario of a train door controller. There is a safety requirement that the door should remain closed whenever the train is moving. However, under certain circumstances such as an emergency, the door must still be opened. The `Door` class, listed in Figure 1, has the following methods: `Open` and `Close` send signals to the door mechanism to release and lock the doors, respectively. `Start` and `Stop` are events monitored by the door, which indicate that the train has started or stopped moving, respectively. `Alarm` is an event which indicates that someone pressed an emergency button and `Safe` is an event indicating that the emergency situation is over.

```
1  public class Door {
2    public bool danger, closed, moving;
3
4    private void Invariant() {
5      Contract.Invariant(danger ? !closed : true);
6    }
7
8    public Door() {
9      closed = true; moving = false; danger = false;
10   }
11   // Controlled operations
12   public void Open() {
13     Contract.Requires(closed && !moving);
14     closed = false;
15   }
16   public void Close() {
17     Contract.Requires(!closed && !danger);
18     closed = true;
19   }
20   // Monitored events
21   public void Start() {
22     Contract.Requires(!moving);
23     moving = true;
24     if (!danger) closed = true;
25   }
26   public void Stop() {
27     Contract.Requires(moving);
28     moving = false;
29   }
30   public void Alarm() {
31     Contract.Requires(!danger);
32     danger = true; closed = false;
33   }
34   public void Safe() {
35     Contract.Requires(danger);
36     danger = false;
37   }
38 }
```

Figure 1: Train door controller

Each of these methods is equipped with a requires clause which prevents it from being executed in the wrong context. However, none has associated ensures clauses.

Given one particular method, understanding if it provides the expected functionality is relatively easy. On the other hand, understanding the interaction between them is a much more complex task: we need to analyze every possible method call sequencing.

In order to assist the developer with this challenging problem, as the first step of our approach we propose the automatic construction of a finite state typestate-like abstraction as the one in Figure 2. Our CONTRACTOR.NET tool automatically built this abstraction in less than 15 seconds on a Core i5 computer with 4 GB of RAM.

Each of the abstract states groups all the class instances enabling the same set of methods, and the initial one is marked with a double circle. For instance, `S2` is the only state in the abstraction that enables `Close`, `Start` and `Alarm`, while disabling `Open`, `Stop` and `Safe`. This instance grouping strategy, which we call *enabledness preserving*, provides a good abstraction size/precision compromise and has proved to be useful to discover inconsistencies between the implementation and the expected global functionality [3].

After the `Door` class is released, other pieces of software might start using it. Figure 3 presents one such possible client. In this scenario the train faces an emergency while moving between stations, but it turns out to be a false alarm.

```
1 public void AlarmScenario() {
2   Door door = new Door();
3   door.Start(); // Departing station
4   door.Alarm(); // Emergency button pressed...
5   door.Safe();  // ... but it was a false alarm
6   door.Stop();  // Arriving at next station
7   door.Open();  // Opening doors
8   door.Close(); // Getting ready to depart again
9 }
```

Figure 3: Train door client code

The client developer might want to statically verify his code using CODE CONTRACTS. Unfortunately, the `Door` class API does not include any ensures clauses, making it impossible for CODE CONTRACTS to successfully verify it.

For the second and final step of our technique, we present a contract strengthening mechanism that enriches the `Door` class CODE CONTRACTS specification with the restrictions obtained from its generated typestate. This strengthened specification makes it easier for the CODE CONTRACTS static verifier to discover an error in the client code. In particular, at line 7 in Figure 3 the client is attempting to open the doors when in fact they were already opened when the emergency started; therefore violating the `Open` method requires.

The problem originates from the wrong assumption that the doors would automatically be closed once the emergency was over (`Safe` method). This misunderstanding of the `Door` functionality could have been avoided by inspecting the typestate in Figure 2. Following the client method calls gives the trace $\texttt{S1} \xrightarrow{\texttt{Start}} \texttt{S3} \xrightarrow{\texttt{Alarm}} \texttt{S5} \xrightarrow{\texttt{Safe}} \texttt{S6} \xrightarrow{\texttt{Stop}} \texttt{S2}$. In this last state the `Open` method is not enabled and, with our enriched specification, this is detected by CODE CONTRACTS.

This simple example shows how our approach can assist client code developers in finding problems with respect to the APIs that they use. It also hints at the potential of using enabledness preserving abstractions to validate contract specifications provided explicitly on a per method basis.

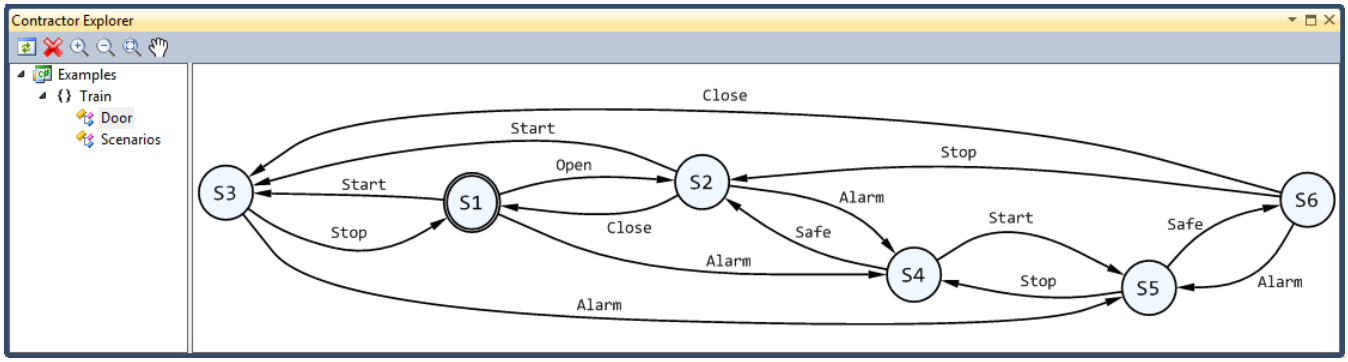

Figure 2: Train door controller abstraction

In the next sections we will elaborate on how to automatically construct typestates such as the one depicted in Figure 2 and how to use them to find faulty clients such as the one in Figure 3.

3. TYPESTATE CONSTRUCTION

Typestates are automatically constructed by setting a fixed level of abstraction, namely *enabledness* of methods [3]. A *class* C can be seen as a structure $\langle M, F, R, \text{inv}, \text{init}\rangle$, where $M = \{m_1, \ldots, m_k\}$ is a finite set of *public method labels*, F is an M-indexed set of *method implementations*, R is an M-indexed set of *requires clauses*, inv is the *class invariant*, and init denotes the possible *initial conditions* given by the constructors. Given a class C and two instances c_1, c_2, we say that c_1 and c_2 are *enabledness equivalent* (noted $c_1 \equiv_e c_2$) iff for every $m \in M$: c_1 satisfies R_m iff c_2 satisfies R_m.

The set of class instances, when quotiented by $\equiv_e$, results in a set of abstract states, such that each one is mapped to a (distinct) group of enabled methods. Each abstract state groups all the instances that share the same set of enabled methods, and can be characterized by a *state invariant*. Formally, the invariant of an abstract state given by a set of methods $ms \subseteq M$ is a function inv_{ms} that takes an instance of C and returns a boolean. It is formally defined as:

$$\text{inv}_{ms}(c) \overset{\text{def}}{\Leftrightarrow} \text{inv}(c) \wedge \bigwedge_{m \in ms} c \text{ safisties } R_m \wedge \bigwedge_{m \notin ms} c \text{ does not satisfy } R_m$$

Having defined the set of abstract states, we still need to figure out which transitions must be added in order to obtain an enabledness-preserving abstraction. In other words, given abstract states ms_1 and ms_2 we must decide if it is possible for a class instance c_1 in ms_1 to execute a method $m \in ms_1$ and evolve into a class instance c_2 in ms_2.

Following the work in [3], where abstractions are built using reachability queries, we now instead use the Code Contracts static verification engine. More concretely, for every method $m \in ms_1$, in order to decide if the transition $ms_1 \xrightarrow{m} ms_2$ needs to be added to the abstraction, we extend class C with an extra method, as depicted in Listing 4.

```
1 private void ms1_m_ms2() {
2   Contract.Requires(this.inv_ms1());
3   Contract.Ensures(!this.inv_ms2());
4   this.m();
5 }
```

Figure 4: A method that checks if $ms_1 \xrightarrow{m} ms_2$

If the ensures clause is successfully verified, then every instance that satisfies the state invariant of ms_1 does *not* satisfy the invariant of ms_2 after executing the method m. Therefore, if the method is verified, the transition $ms_1 \xrightarrow{m} ms_2$ is not added to the abstraction. On the other hand, if the method is not verified, or if the static verifier is uncertain, the transition is added[2].

The reader may notice that a class with k public methods has potentially 2^k reachable abstract states. A naïf implementation would compute all the 2^k states and its transitions, only to later restrict the result to the reachable fragment. In [3] we present an algorithm that performs a parallel BFS exploration of the abstract state space, therefore drastically reducing running times.

The obtained typestate has multiple uses: *i)* It can be used for class validation [3]. That is, for comparing the typestate with the developer understanding of the class intended functionality and find if there are any discordances (i.e. potential bugs). *ii)* It can also be used to enforce correct client usage. That means checking that client sequences are included in the set of traces of the class typestate. This enforcement can be performed statically via typestate checking (e.g., [4]) or by *typestate monitoring* using approaches such as [2].

4. CONTRACT STRENGTHENING

The second step of our approach consists in leveraging on the inferred typestate to verify that clients use the class correctly. This is accomplished by extending the original Code Contracts specification of the API to add restrictions that make it adhere to its underlying typestate. The approach we follow is that of *typestate monitoring* (e.g., [2]): instrumenting the class in order to raise an exception when a client violates the typestate. In our setting we work with classes that use Code Contracts clauses and, instead of run-time checks, we add requires and ensures clauses.

For instance, consider the `Close` method from Figure 1. In Figure 5 we show its strengthened version, employing the information extracted from the typestate in Figure 2. Line 2 defines a new variable to keep track of the current abstract state. Line 5 is the requires clause from the original class. Since `Close` is only enabled on states `S2` and `S6`, line 6 codifies this restriction using the newly added variable. Line 8 specifies how the state variable is updated, which depends on its previous value. Lines 9 and 10 do the actual state variable update. Finally, line 11 comes from the original class.

[2]If method m has parameters the encoding is trickier; see [3].

```
1  public class Door {
2    public int state;
3    // ...
4    public void Close() {
5      Contract.Requires(!closed && !danger);
6      Contract.Requires(state==2 || state==6);
7      Contract.Ensures(old(state)==2 ?
8                       state==1 : state==3);
9      if (state==2) state = 1;
10     else if (state==6) state = 3;
11     closed = true;
12   }
13 }
```

Figure 5: Strengthened `Close` method

Notice how not only we updated the state variable, but also we added an ensures clause with this information.Code Contracts can use this clause to perform modular static verification of client usage of the `Door` class. Code Contracts can be used as usual without the need to make it typestate-aware or modify it whatsoever. Client code can be checked at compile-time using the static analysis engine. Users which turn off the static checker, or if the verification does not provide a definite answer (due to potential false positives), will still enjoy run-time specification checks.

In Figure 5 the `state` variable update only depends on its previous value since the typestate was a deterministic automaton. In the presence of non determinism, the `state` variable update will also depend on some of the class fields.

5. PLUG-IN DETAILS

As we mentioned before, Contractor.NET is a Visual Studio Extension. Users are presented with a *Contractor Explorer* tool window (shown in Figure 2), which allows them to choose which class to analyze.

Once the desired class is selected, the analysis starts in the background. The typestate view is automatically updated every time a new state or transition is found so that the user can see a preliminary result without having to wait for the analysis to complete.

The resulting typestate is shown on the right panel (using the Microsoft MsAgl library[3]), where the user can select and rearrange the layout. Features like zooming and panning are also provided. Hovering the mouse over an abstract state provides a tooltip indicating the set of enabled actions. Finally the user can export the typestate in many formats, including XML and scalable vector graphics.

Our current version of Contractor.NET, which is available for download, implements all the features presented in this paper. That is, the automatic typestate inference and contract strengthening, as explained in §3 and §4.

Finally, Contractor.NET works at Common Intermediate Language (CIL) level (via Microsoft's CCI tool[4]), so it can handle classes written in any .NET language.

6. RELATED WORK

Due to space limitations we will briefly discuss the approaches which are closest to ours.

Regarding typestate construction, our technique is related to approaches that statically synthesize safe client interfaces (e.g., [6]) out of a program. Any sequence of methods that is not accepted by our abstraction will not be allowed by a program. However, interface synthesis approaches aim to obtain a set of safe traces from a client perspective, using abstraction for verification purposes [4, 1] rather than validation. The models they construct tend to be overly restrictive and not suitable for human-inspection.

From a client verification perspective, the models of [6] may be too conservative and rule out legal usages. In contrast, our abstractions over-approximate the class state space, so we might accept some invalid client sequences. Overall, considering our technique builds up a typestate from scratch, we leave developers in a better position since they can reject more invalid clients than before.

Regarding contract strengthening using typestate information, the problem is related to typestate run-time monitoring [2]. The closest approach to our work is [7]: a Java Modeling Language (JML) extension to include explicit typestate annotations. These are then translated to regular JML annotations following a strategy similar to ours.

To the best out knowledge ours is the first approach to deal with both typestate construction and enforcement via contract strengthening.

7. CONCLUSION

In this work we presented Contractor.NET, a Visual Studio extension which allows developers to automatically construct a typestate for the class they are developing. This inferred typestate is enabledness-preserving, a level of abstraction which conveys a concise, yet representative, view of the class state space; assisting the programmer in understanding the code and finding problems in it. This abstraction can also be used to strengthen the class contract specification, enabling static verification and run-time checking of client code via Code Contracts.

For future work, we plan to finish the implementation the automatic contract strengthening, particularly considering the necessary changes in order to support non-deterministic typestates. Once this is done, we plan to experiment with our technique in more .NET classes, particularly the ones from its standard class libraries. Finally, we are also working in a multithreaded version of the algorithm (see [3]).

8. REFERENCES

[1] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02*, pages 1–3, 2002.

[2] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *FSE '08*, pages 36–47, 2008.

[3] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Program abstractions for behaviour validation. In *ICSE 2011 (to be published)*, 2011.

[4] R. DeLine and M. Fahndrich. Typestates for objects. *ECOOP'04 (LNCS)*, pages 465–490, 2004.

[5] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC'10*, pages 2103–2110, 2010.

[6] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *FSE '05*, pages 31–40, 2005.

[7] T. Kim, K. Bierhoff, J. Aldrich, and S. Kang. Typestate protocol specification in JML. In *SAVCBS '09*, pages 11–18. ACM, 2009.

[3] `http://research.microsoft.com/en-us/projects/msagl`

[4] `http://cciast.codeplex.com`