TrueChange™ under the hood: how we check the consistency of large models (almost) instantly

Hugo Lourenço
Principal Software Engineer, OutSystems
hugo.lourenco@outsystems.com

Rui Eugénio
Lead Software Engineer, OutSystems
rui.eugenio@outsystems.com

***Abstract*—The OutSystems Platform is a visual model-driven development and delivery platform that allows developers to create enterprise-grade web and mobile applications.**

The models created with the platform are translated by its compiler into a set of standard-technology artifacts (C#, JavaScript, SQL, etc). The model must be checked for consistency (i.e., that it is well-formed and well-typed) before compilation can proceed. Our Integrated Development Environment (IDE) does this in real-time: after each change made a developer, the IDE either automatically *heals* the other parts of the model that are impacted by the change, or provides immediate feedback on the errors that must be manually corrected.

It is not uncommon for an OutSystems model to contain in excess of 200,000 individual elements. Handling large models efficiently is thus of paramount importance: consistency checks must run as fast as possible, otherwise the developer's experience is significantly impaired.

In this paper we present the techniques we have developed to speed up consistency checks, and which resulted in the TrueChange™ engine. We use an incremental approached paired with automatically managed back pointers. We believe these techniques are of general application and not limited to our particular case.

***Index Terms*—consistency check, large models, model driven development, domain specific language**

I. Introduction

The OutSystems Platform [5] is a visual model-driven development and delivery platform that allows developers to create enterprise-grade web and mobile applications. Using Service Studio, the platform's IDE, it is possible to design in a single place all the aspects of an application, including, among others, its user interfaces, business logic, database models, and integration with external systems (e.g., via REST services). Each of these aspects is developed using its own (type-safe) domain specific language. An OutSystems model can thus be seen as consisting of several interdependent sub-models.

The OutSystems Platform is used by its clients to create very large applications. OutSystems promotes and encourages the use of Agile methodologies [1], and consequently it is extremely important to allow developers to be highly productive and efficient not only when *creating* applications but also, and more importantly, when *changing* them to adapt to new business needs.

This need is addressed in two main ways. Firstly, the OutSystems language is strongly typed. As it is well known, using a strongly typed language makes changing applications safer and faster, as a significant amount of errors can be detected and forcibly fixed at design-time rather than at run-time. A simple example is changing a function's input parameter from optional to mandatory. All function calls will be in a state of error if the corresponding argument's value is not filled in.

Secondly, the model is inspected for possible inconsistencies. These are not necessarily *errors*, but are nonetheless issues that the developer must act upon because they may for instance impact the usability of the application. An example of this is warning developers if they place too many remote calls in an application's startup method, or if the maximum length of an input field in a form doesn't match the maximum length of the corresponding database entity attribute.

The TrueChange™ engine is the component of Service Studio that inspects and fixes the model in real-time. When a developer changes the model, TrueChange™ provides immediate feedback on the impact of the change and, in many cases, it is able to automatically apply corrective measures. As applications created with the OutSystems Platform can be very large (models containing more than 200,000 elements are not uncommon - see Table I), having a high performant TrueChange™ was a top priority.

Our main approach is to run the engine incrementally, analyzing only the parts of the model that have changed or are known to be affected by a change. We call the latter the *verification dependencies*. E.g., the verify dependencies of an input parameter includes all arguments referring to it: whenever the parameter's type changes, or when an optional parameter is made mandatory, we need to *re-check* all those arguments. TrueChange™ speed is dependent on calculating verification dependencies quickly and efficiently. In this paper we'll present the data structure we designed to make this possible.

The remainder of the paper is organized as follows. In section II we briefly introduce the OutSystems language and the internal representation of our models. The TrueChange™ engine is presented in section III. We conclude the paper by evaluating the impact of TrueChange™ in section IV and with some final remarks.

II. The OutSystems language

The OutSystems language [4] is a visual language. Developers use Service Studio, the OutSystems Platform IDE, to visually design all the layers of their web or mobile applications, and then publish them to the OutSystems Platform

server (see Figure 1). The OutSystems compiler generates all the required artifacts (C#, JavaScript, SQL scripts, etc) and deploys them to the application server and database. At that point the application can be accessed by the end-users through their devices.

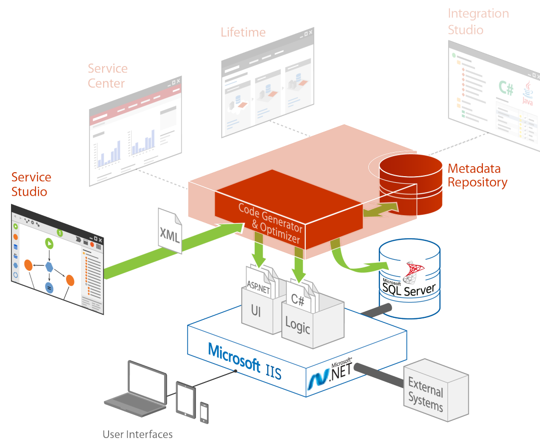

Fig. 1. The OutSystems Platform

In this paper, and for simplicity, we'll focus our attention on a subset of the OutSystems language consisting of *Actions* [3]. Actions are used to design business logic, and are very similar to methods in textual languages such as C# or Java. The OutSystems language supports several different kinds of actions, including client actions (which run in the client device, be it a smartphone, tablet, or browser) and server actions (which run in the server).

In Figure 2 we illustrate Service Studio's user interface. The *Logic* tab is selected and action *ProcessRequest* is being edited. Actions are represented as directed graphs. The nodes that can be used to create an action are depicted in the toolbox on the left. In our example, action *ProcessRequest* consists of three nodes: *Start*, *SendEmail*, and *End*. *Start* and *End* have the expected role: they mark the beginning and end of the action, respectively. *SendMail* is a *Run Server Action* node. These nodes are used to invoke (call) another action. In this case we're invoking action *SendEmail* which, as can be seen in the tree to the right, is defined in the same application. This action expects a single input parameter, *EmailAddress*. The parameter is currently selected in the tree, and thus its properties are being displayed in the box below the tree. As we can see, the parameter is of type *Email* and is mandatory.

A. Meta-model

OutSystems models are persisted and transported as binary XML files. In memory (both when being edited in Service Studio or while being processed by the platform's compiler), a model is represented as an objects graph.

Although OutSystems supports and develops a particular concrete language, we designed the language from the ground up to be easily extensible. The language's meta-model is represented as a XML file, from which we generate the model classes. In Listing 1 we present an abridged and simplified version of the meta-model used to specify *Actions*. For the sake of completeness, an equally abridged and simplified version of the meta-model's meta-model is presented in Listing 2.

```
1  <MetaModel xmlns:xsi="http://www.w3.org/2001/
       XMLSchema-instance"
2    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    xsi:schemaLocation="http://www.outsystems.com
         MetaModel.xsd">
4
5   <Class name="ESpace">
6     <Property name="Name" type="Text" />
7     <Child name="Actions" type="Action" />
8   </Class>
9
10  <Class name="Action">
11    <Property name="Name" type="Text" />
12    <Child name="InputParameters" type="
        InputParameter" />
13    <Child name="Nodes" type="ActionNode" />
14  </Class>
15
16  <Class name="InputParameter" >
17    <Property name="Name" type="Text" />
18    <Property name="Type" type="Type" />
19    <Property name="IsMandatory" type="Bool" />
20  </Class>
21
22  <Class name="ActionNode">
23    <Property name="Target" type ="ActionNode" />
24  </Class>
25
26  <Class name="Start" base="ActionNode" />
27  <Class name="End" base="ActionNode" />
28
29  <Class name="Execute" base="ActionNode">
30    <Property name="Action" type="Action" />
31    <Child name="Arguments" type="Argument" />
32  </Class>
33
34  <Class name="Argument" verifyDependencies="
      Parameter.IsMandatory, Parameter.Type">
35    <Property name="Parameter" type="
        InputParameter" />
36    <Property name="Value" type="Expression"
        isOptional="true" />
37  </Class>
38 </MetaModel>
```

Listing 1. Meta-model for Actions

```
1  <xs:schema elementFormDefault="qualified" xmlns:
       xs="http://www.w3.org/2001/XMLSchema">
2   <xs:element name="MetaModel">
3     <xs:complexType>
4       <xs:choice minOccurs="1"
5                  maxOccurs="unbounded">
6         <xs:element name="Class" type="Class" />
7       </xs:choice>
8     </xs:complexType>
9   </xs:element>
10
11  <xs:complexType name="Class">
12    <xs:choice minOccurs="0"
13               maxOccurs="unbounded">
14      <xs:element name="Property" type="Prop" />
15      <xs:element name="Child" type="Child" />
16    </xs:choice>
17    <xs:attribute name="name" type="xs:string"
        use="required" />
```

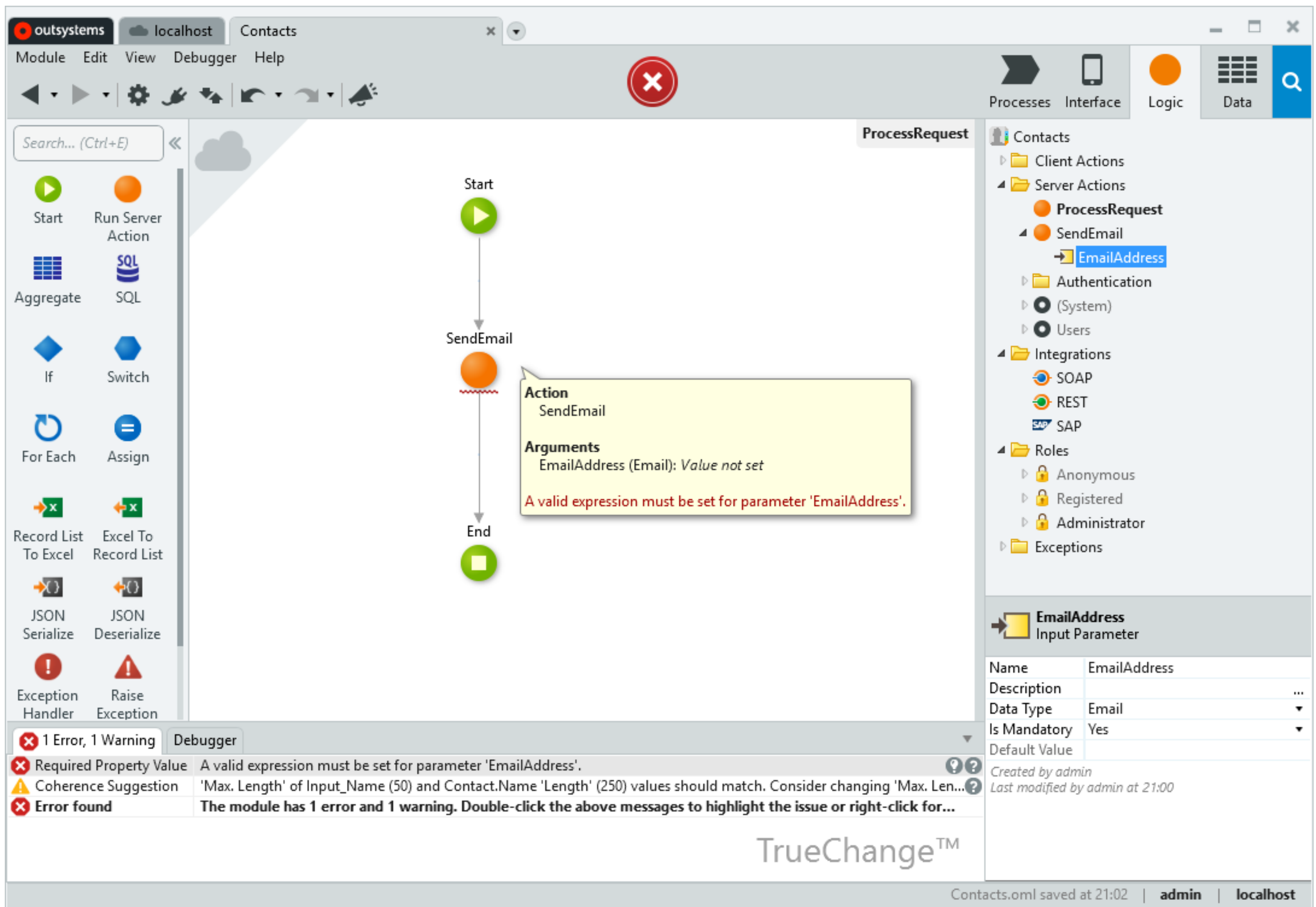

Fig. 2. Service Studio interface. Action *ProcessRequest* is being edited and parameter *SendEmail.EmailAddress* is selected in the tree.

```
18    <xs:attribute name="base" type="xs:string" />
19    <xs:attribute name="verifyDependencies"
20                  type="xs:string" />
21  </xs:complexType>
22
23  <xs:complexType name="Prop">
24    <xs:attribute name="name" type="xs:string"
25                  use="required" />
26    <xs:attribute name="type" type="xs:string"
27                  use="required" />
28    <!-- a missing value is interpreted as "false
        " -->
29    <xs:attribute name="isOptional" type="xs:
        boolean" />
30  </xs:complexType>
31
32  <xs:complexType name="Child">
33    <xs:attribute name="name" type="xs:string"
        use="required" />
34    <xs:attribute name="type" type="xs:string"
        use="required" />
35    <!-- a missing value is interpreted as "false
        " -->
36    <xs:attribute name="isSingleton" type="xs:
        boolean" />
37  </xs:complexType>
38 </xs:schema>
```

Listing 2. Meta-model for the meta-model

Listing 1 should be easy to follow (some of the attributes, namely *verifyDependencies* in line 34, will be discussed in section III). We're declaring several classes, including *ESpace*, which corresponds to an application module, and *Action*. For each class we declare their *Properties* and *Children*. In UML terminology, properties are *attributes* and children represent *aggregation* relations. For instance, class Action has, or is made up of, a set of *InputParameter* and a set of *ActionNode* (lines 12 and 13 in Listing 1). The *Execute* class represents the *Run Server Action* node discussed above. It declares a property, *Action*, to store the action being executed, and a collection of *Argument*. An *Argument* refers a particular parameter and provides a value for it. The classes declared in the meta-model can be used as the type for properties and collections in other classes.

B. Model classes

The meta-model is not directly used in its XML form. Instead, both Service Studio and the compiler use C# classes that we generate from the meta-model. The generated model classes include a substantial amount of predefined behavior, such as code for load, save, copy/paste, merge, and verify. In Listing 3 we illustrate a bare-bones version of the generated

classes that, for compactness reasons, doesn't include any of the behavior just enumerated.

Since the meta-model is an input to the model classes generator, it is possible to define a completely new language by defining its corresponding meta-model. All of these potential languages share some common characteristics, and depend on a set of non-generated classes that are referred to in Listing 3. These include class *Type* for types, class *Expression* for expressions, and class *ModelObject* class, which is the base class for all model classes. Model class instances store an auto-generated id that uniquely identifies the instance inside a particular model.

```
1 using System.Collections.Generic;
2 using OutSystems.BaseModel;
3
4 namespace OutSystems.Model {
5
6  partial class ESpace : ModelObject {
7   public string Name { get; set; }
8   public List<Action> Actions { get; };
9  }
10
11 partial class Action : ModelObject {
12  public string Name { get; set; }
13  public List<InputParameter> InputParameters {
        get; };
14  public List<ActionNode> Nodes { get; };
15 }
16
17 partial class InputParameter : ModelObject {
18  public string Name { get; set; }
19  public Type Type { get; set; }
20  public bool IsMandatory { get; set; }
21 }
22
23 partial class ActionNode : ModelObject {
24  public ActionNode Target { get; set; }
25 }
26
27 partial class Start : ActionNode { }
28
29 partial class End : ActionNode { }
30
31 partial class Execute : ActionNode {
32  public Action Action { get; set; }
33  public List<Argument> Arguments { get; };
34 }
35
36 partial class Argument : ModelObject {
37  public InputParameter Parameter { get; set; }
38  public Expression Value { get; set; }
39 }
40 }
```

Listing 3. Generated model classes

C. Models

As mentioned above, an OutSystems model is represented in memory as an object graph whose entry-point (the main object) is an instance of class *ESpace*, and is persisted as a binary XML file. It should come as no surprise that a model's XML representation is a *serialization* of the objects graph.

Consider the example illustrated in Figure 2, which consists of actions *SendEmail* and *ProcessRequest*. The corresponding model is presented in Listing 4. For simplicity and compactness, action *SendEmail* has no nodes. As you can easily check, this model conforms to the meta-model presented in Listing 1. The representation for object references consists of the target object's type and its id. For instance, in line 12 the value for the *action* attribute is interpreted as the instance of the *Action* class with id 4, which correspond to action *SendEmail* in line 4 [1]. Notice that in this model we're not providing a value for the argument in line 14. As we'll see later on, this will be detected as an error, since the corresponding parameter (*InputParameter:5*, line 6) is declared to be mandatory.

```
1 <Model>
2  <ESpace id="1" name="Contacts">
3   <Actions>
4    <Action id="4" name="SendEmail">
5     <InputParameters>
6      <InputParameter id="5" isMandatory="true"
           name="EmailAddress" type="Type:3" />
7     </InputParameters>
8    </Action>
9    <Action id="6" name="ProcessRequest">
10    <Nodes>
11     <Start id="7" target="Execute:8" />
12     <Execute action="Action:4" id="8" target=
          "End:9">
13      <Arguments>
14       <Argument id="10" parameter="
            InputParameter:5" />
15      </Arguments>
16     </Execute>
17     <End id="9" />
18    </Nodes>
19   </Action>
20  </Actions>
21 </ESpace>
22 </Model>
```

Listing 4. Model example

III. TrueChange™

TrueChange™ is the sub-component of Service Studio responsible for checking the model for consistency. Part of its work is akin to syntactic and semantic analysis in traditional languages: it checks if the model is well-formed (e.g, do all *Start* nodes have exactly one outgoing arrow) and well-typed (e.g., are all action arguments of a type compatible with their corresponding argument)[2]. It also detects and warns about many other problems, including potential performance issues.

Most of the work is actually delegated on the model classes, and TrueChange™ acts as a "supervisor" or "bookkeeper". A substantial part of the work is carried out by code that is generated from the meta-model. This includes, e.g., handling mandatory vs optional properties, value ranges, collections cardinality, etc. For the cases not covered by the model classes generator we can provide a manual implementation. We thus have two approaches in what concerns specifying consistency checks: we can *declare* them in the meta-model, or *code* them.

An example of a manually coded consistency check is presented in Listing 5, where we present the verify code for the *Argument* class. This code validates that a value is provided

[1]The actual representation used by the *real* OutSystems model is richer and more complex, but that is outside the scope of this paper.

[2]This is akin to syntactic and semantic checking in textual languages.

if the parameter is mandatory, and that when a value is set it must be compatible with the parameter's type. In the case of the model in Listing 4, calling *CalculateVerifyMessages* on action *ProcessRequest* would return the following message:

Error: A valid expression must be set for parameter 'EmailAddress'.

```
using OutSystems.BaseModel;
using System.Collections.Generic;

namespace OutSystems.Model {
  partial class Argument {

    protected override IEnumerable<VerifyMessage>
        CalculateVerifyMessages() {
      if (Parameter.IsMandatory && Value == null){
        yield return new VerifyMessage() {
          Kind = VerifyMessageKind.Error,
          Text = $"A valid expression must be set
              for parameter '{Parameter.Name}'."
        };
      }

      if (Value != null && !Value.Type.
          IsConvertibleTo(Parameter.Type)) {
        yield return new VerifyMessage() {
          Kind = VerifyMessageKind.Error,
          Text = $"'{Parameter.Type.Name}' required
              instead of '{Value.Type.Name}'."
        };
      }
    }
  }
}
```

Listing 5. Verify code for class *Argument*

A. Incremental verification

It is not feasible to check the whole model every time anything changes. We use a *cache* mechanism instead. In Listing 6 we present an abridged and simplified version of the code that powers TrueChange™. For each object we keep track of its verify messages (field *verifyMessages*, line 18) and if these are up to date (field *isVerified*, line 19). The *Verify* method returns the object's verify messages, calling *CalculateVerifyMessages* if needed to calculate them.

```
using System.Collections.Generic;
using System.Linq;

namespace OutSystems.BaseModel {
  enum VerifyMessageKind {
    Warning,
    Error
  }

  class VerifyMessage {
    public VerifyMessageKind Kind;
    public string Text;

    public override string ToString() => $"{Kind
        }: {Text}";
  }

  partial class ModelObject {
    VerifyMessage[] verifyMessages;
    protected bool isVerified = false;

    List<ModelObject> referers = new List<
        ModelObject>();

    public IEnumerable<VerifyMessage> Verify() {
      if (!isVerified) {
        verifyMessages = CalculateVerifyMessages()
            .ToArray();
        isVerified = true;
      }
      return verifyMessages;
    }

    protected virtual IEnumerable<VerifyMessage>
        CalculateVerifyMessages() {
      yield break;
    }

    protected void InvalidateReferers() {
      foreach (var obj in referers) {
        obj.isVerified = false;
      }
    }

    // called when a property is being changed
        from "oldValue" to "newValue"
    protected void UpdateReferers(ModelObject
        oldValue, ModelObject newValue) {
      if (oldValue != null) {
        oldValue.referers.Remove(this);
      }
      if (newValue != null) {
        newValue.referers.Add(this);
      }
    }
  }
}
```

Listing 6. TrueChange™ basic functionality

The general rule for a model object is that if any of its properties or collections changes then it needs to be re-verified [3]. The model classes generator includes the code necessary to set *isVerified* to false when changing the model. A subset of such code is presented in Listing 7 for the properties of classes *InputParameter* and *Argument* (this is a more detailed version of the code previously presented in Listing 3). As you can see, whenever a property is changed we set *isVerified* to false.

In subsection III-B we present and discuss the fields and methods that we skipped over both in Listing 6 and Listing 7.

```
using OutSystems.BaseModel;

namespace OutSystems.Model {

  partial class InputParameter : ModelObject {
    // Name and Type properties ommitted
    // for brevity

    private bool _isMandatory;
    public bool IsMandatory {
      get => _isMandatory;
      set {
        if (_isMandatory != value) {
          InvalidateReferers();
          _isMandatory = value;
          isVerified = false;
        }
      }
    }
```

[3] We're over-simplifying given the limited scope of this paper.

```
20      }
21
22    partial class Argument : ModelObject {
23      private InputParameter _parameter;
24      public InputParameter Parameter {
25        get => _parameter;
26        set {
27          if (_parameter != value) {
28            UpdateReferers(_parameter, value);
29            _parameter = value;
30            isVerified = false;
31          }
32        }
33      }
34
35      private Expression _value;
36      public Expression Value {
37        get => _value;
38        set {
39          if (_value != value) {
40            UpdateReferers(_value, value);
41            _value = value;
42            isVerified = false;
43          }
44        }
45      }
46    }
47  }
```

Listing 7. Generated code for tracking model changes

B. Referers

At this point there's still a puzzle piece missing: how can we efficiently *propagate* the effects of a local model change? As we discussed previously, our model in Listing 4 is in a state of error: the argument for the mandatory parameter *EmailAddress* is not filled in. There are two ways in which the developer can make this error disappear:

- By acting locally in the argument, setting the value of property *Expression*. This is readily detected by TrueChange™, since the argument's own *isVerified* is set to false by the setter for property *Expression*.
- By acting non-locally in the parameter, making it optional. For incremental verification to work in this case we need to ensure that *all* arguments referring to the parameter are marked as needing validation (i.e., that we set their *isVerified* field to false).

The purpose of the *referers* structure that we skipped over on the previous listings is optimizing the second case. For each model object we keep the list of the other model objects that refer to it through their properties. This is vaguely similar to the purpose of the HTTP *referer* header [2] (the misspelling in our case is coincidental): this optional header identifies the URL of the web page that links to resource being requested.

Referrers can be seen as back-pointers. By keeping them we can very quickly navigate back in the model. Our model classes generator takes advantage of information provided in the meta-model to include additional operations in the property setters (see Listing 7):

- When setting a property which refers to another model object we update its *referers* list (lines 28 and 40, and method *UpdateReferers* in Listing 6). Notice that this means that *referers* are managed in a completed automated way and pose no burden on the teams developing Service Studio.
- When setting a property that corresponds to a verifcation dependency (specified through the *verifyDependencies* attribute in the meta-model) we invalidate all the referring objects (line 14 and method *InvalidateReferers* in Listing 6).

In our meta-model (Listing 1, line 34) we've specified that class Argument has two verifaction dependencies: *Parameter.IsMandatory* and *Parameter.Type*. With this we're stating that whenever the properties *IsMandatory* or *Type* of the input parameter referred through the argument's *Parameter* property change, then we need to re-verify the argument. This is aligned with the manual code in Listing 5, in which we indeed do access these properties. Note that the manually written code has an implicit "cache invalidation" criteria: the verify messages that we calculated for an argument are (potentially) no longer applicable whenever either the *IsMandatory* or *Type* properties of its *Parameter* are changed.

By explicitly declaring the dependency in the meta-model we're providing valuable information for the model classes generator. Namely, we now know that there are some model objects that need to be *notified* of changes to properties *IsMandatory* and *Type*. In our example, this results in the call to *InvalidateReferers* in line 14 of Listing 7. Notice that the verification dependency is declared in class *Argument*, but the code is added to class *InputParameter*.

Ideally, verification dependencies would be inferred automatically - a very interesting line of research on its own. In order to make that possible we would need a declarative way of specifying verification code. In our example, the code in Listing 5 was written by hand, so there isn't an easy way to infer the verification dependencies from this code. Pragmatically, when developing Service Studio we either specify the verification dependencies in the meta-model or explicitly invalidate the objects known to be affected by a change.

For the base model, namely for types and expressions, we have automatic verification dependencies. For instance, properties of type *Type* are treated as a special case. Whenever a property of type *Type* in a particular model object changes, all expressions referencing the object are invalidated.

The referers structure is persisted together with the model. For our example in Listing 4 we'll store the information presented in Listing 8. In line 3, for instance, we're storing the information that the object with id 5 (the input parameter *EmailAddress*) is referred by the argument argument with id 10 (the argument in action *ProcessRequest*).

Notice that the referers structure is needed to solve the general case of quickly determining the model objects that may be impacted by a change elsewhere in the model. It is true that we can easily identify the model objects that are in a state of error, and so it could be argued that after a change we would simply re-process those objects. But of course this doesn't cover the case of model objects that *will be* in a state

of error only as consequence of a change. In our example, if our input parameter was optional the model would be OK. After making the input parameter mandatory we need to get an error in the argument.

```
<Referers>
  <Object id="4" referers="Execute:8" />
  <Object id="5" referers="Argument:10" />
  <Object id="8" referers="Start:7" />
  <Object id="9" referers="Execute:8" />
</Referers>
```

Listing 8. Referers

IV. Evaluation

A. Time requirements

In order to assess the benefits of our strategy we looked at a set of application models created with the OutSystems platform (Table I) and measured the time taken to do a full consistency check of the complete model (Full verify time) versus an incremental check (Incremental verify) for a representative set of model change.

One of the models, Service Center, is a web management console for the OutSystems Platform. The remaining models are part of real applications created by OutSystems' clients that kindly allowed us to use them for analysis purposes, provided that we obfuscated the model names. These models are some of the largest in terms of number of elements, and were selected precisely because of that characteristic.

The tests were conducted using a Dell laptop with a 2.6GHZ (3.50 GHz max) Intel Core i7-6700HQ CPU, 16GB of RAM, and an Intel SATA SSD disk drive. For each model we measured the time taken to do a complete consistency check and the time taken by an incremental check for several different kinds of changes. The elements that we changed were selected among the ones with the highest amount of usages inside the model, in order to further stress the algorithm - the time TrueChange™ takes depends primarily on the number of *referers* of the element being changed. The kind of element determined the change operation, as not all operations make sense for every type of element. For instance, Model #1 has a high number of user interface screens but low entity count.

Table I summarizes the obtained results. As it can be seen, the incremental check times are drastically lower than the full check times. Even in the worst recorded case, TrueChange™ took only 2.4% of the full consistency check time. For our sample, incremental check took 94ms on average. This value is aligned with the needs of using the IDE interactively, i.e. it means that there is low impact on the developer's experience.

Notice that this sample focused on large models and high impact changes, and thus is biased towards worst case scenarios. Consequently, it is easy to conclude that in general the time spent in incremental consistency checks is rather small.

B. Space requirements

We also measure the impact of the *referers* structure in storage space. As can be seen in Table II, the impact is far from negligible, amounting to around one third of the compressed model size. In this table we present, for each model, its size on disk and the size of the *referers* structure.

Even though the impact is high in relative terms, in absolute terms it is acceptable given the low cost of storage. If additionally we take into consideration the considerable performance gains, this is a cost that we feel comfortable paying.

V. Conclusion

In this paper we presented the TrueChange™ engine and how it handles with large models. We focused on how TrueChange™ minimizes the time to do a consistency check after a model change, which is accomplished by doing incremental checks accelerated by the *referers* structure. This structure is a list kept by each element of the other elements in the model that reference it.

The *referers* structure allows to back-navigate in the model very quickly. This is particularly important when propagating the impact of a model change. The benefits of the *referers* structure are not limited to consistency checks, however. We also use it, for instance, to power the *Find Usages* operation, making it extremely fast even for large models.

Using the *referers* structure has a non-negligible impact on the size of the model. However, the gains in incremental consistency check times far outweigh the size increase.

For OutSystems, the major shortcoming of TrueChange™ is the burden placed on the team developing Service Studio. As shown in section III, developers have to guarantee that TrueChange™ is aware of all verification dependencies, either by declaring them in the meta-model or by manually invalidating the objects affected by a particular change. Failing to do so may result in inconsistent models and require a full consistency check. This is just another instance of the general (and difficult) problem of cache invalidation.

This shortcoming provides the inspiration for interesting lines of research that, unfortunately, we haven't yet had time to pursue. These could include, among others:

- use a declarative approach to specifying verify checks for model classes. From these we could generate the verify code and determine the verification dependencies.
- keep using an imperative approach, but collect verification dependencies automatically at runtime. While running the verify code we could keep track of all properties that are read. If any of these changes afterwards then the verify code needs to run again.

Incremental analysis of source code is not a novel idea and is the subject of active research [6], [7]. The techniques presented in this paper are in no way restricted to the particular case of the OutSystems language. We are confident of its generality and feel that it will be useful to the MODELS community. We hope that it will be a good starting point for further discussion during the workshops.

This works has been carried out in an industrial context, and as such it is lacking in certain areas that are typically addressed in academic scenarios. Due to many constraints, of which quick time to market is probably the most important one, researching and exploring related work needed to be

TABLE I
TRUECHANGE™ TIMES

Model	Number of elements	Full check time (ms)	Incremental check		
			Change kind	**Time (ms)**	**% of full check time**
Model #1	259759	11468	Add input parameter to screen	70	0.6%
			Change input parameter type	148	1.3%
			Change input parameter to mandatory	54	0.5%
Model #2	173067	6095	Add attribute to structure	12	0.2%
			Change attribute type	144	2.4%
			Add input parameter to action	88	1.4%
			Change input parameter type	119	2.0%
			Change input parameter to mandatory	83	1.4%
Model #3	211588	6728	Add attribute to entity	5	0.1%
			Change attribute type	18	0.3%
Model #4	194844	6609	Add attribute to structure	9	0.1%
			Change attribute type	9	0.1%
Service Center	278187	26388	Add attribute to entity	215	0.8%
			Change attribute type	388	1.5%
			Add input parameter to action	91	0.3%
			Change input parameter type	77	0.3%
			Change input parameter to mandatory	62	0.2%
Average				**94**	**0.8%**

TABLE II
SIZE OF THE *referers* STRUCTURE

Model	Compressed model size (MB)	Compressed referers size (MB)	Referers size / model size
Model #1	12	4.2	35%
Model #2	7.4	2.8	37%
Model #3	8.3	3.0	36%
Model #4	7.3	2.7	37%
Service Center	16.4	4.5	28%
Average			**35%**

conducted in a limited and time-boxed way. The research that we *did* conduct led us to the decision to design and implement in-house the mechanisms presented in this paper, instead of using existing approaches. The existing work that we find didn't exactly fit our needs or was too hard to integrate into our product.

REFERENCES

[1] Jim Highsmith and Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[2] Henrik Frystyk Nielsen, Roy T. Fielding, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, May 1996.

[3] OutSystems. Implement Application Logic. https://success.outsystems.com/Documentation/11/Developing_an_Application/Implement_Application_Logic/, 2019.

[4] OutSystems. OutSystems 11 Docs. https://success.outsystems.com/Documentation/11, 2019.

[5] OutSystems. Platform Overview. https://www.outsystems.com/platform/, 2019.

[6] Tamas Szabo, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in datalog. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[7] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: A dsl for the definition of incremental program analyses. In D. Lo, editor, *ASE 2016 Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 320–331, United States, 9 2016. Association for Computing Machinery (ACM).