

# Static type-checking for programs developed on the platform 1C:Enterprise

Alexander S. Balyuk<sup>1</sup>, Victoria A. Popova<sup>1</sup>

<sup>1</sup>*Irkutsk State University, 1, K. Marx st., Irkutsk, 664003, Russian Federation*

## Abstract

To improve the quality of software systems, tools are currently being created to eliminate a large number of errors at the development stage. One of the problems that affects the quality of the programs being created is the occurrence of type mismatch errors. Such errors can occur in all programming languages. But for programming languages with static type-checking problems of type inconsistencies are detected even at the stage of compiling the program, whereas for languages with dynamic type-checking, errors of this kind can be detected only during the execution of the program code. For some programming languages with dynamic type-checking, for example, JavaScript and Python, there are already tools for tracking errors. But to date, there is no such tool for the programming language of 1C:Enterprise. The article describes the process of creating a software package for finding type mismatch errors in programs written in the programming language of 1C:Enterprise. The composition of the type system of 1C:Enterprise is described. The types of errors that were found using the developed static verification tool are listed.

## Keywords

static type-checking, dynamic type-checking, type system of 1C:Enterprise, format Configuration Types Tree, static type-checking tool

## 1. Introduction

The rapid development of information technologies makes it possible to automate various production processes more and more every year. Due to the need to constantly introduce new functionality into software products, the risk of errors increases, which is due by the complexity of program code, which becomes more difficult for developers to maintain. Often defects are not detected at the development stage and therefore are present in the released versions of software products. In this case, errors occur when users use the program, which can violate the integrity of the data and suspend work for processes indefinitely.

To improve the quality of programs, tools are currently being created that allow you to eliminate a large number of errors even at the stage of creating software systems. Such systems include configurations developed on the platform 1C:Enterprise, in which a significant part of the errors are associated with dynamic type-checking.

Tools for detecting type mismatch errors have already been developed for some programming languages with dynamic dynamic type-checking. For example, Google has created such a type

---


*Information Technologies: Algorithms, Models, Systems (ITAMS) 2021*

✉ [sacha@hotmail.ru](mailto:sacha@hotmail.ru) (A. S. Balyuk); [victorypopova1@gmail.com](mailto:victorypopova1@gmail.com) (V. A. Popova)

🆔 0000-0001-9483-7108 (A. S. Balyuk); 0000-0002-7764-1995 (V. A. Popova)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

checking tool for the programming language of JavaScript as part of the Closure Compiler project.

The process of developing a static type-checking tool for 1C:Enterprise is characterized by complexity due to a sufficiently large number of data types and methods that are provided as a built-in tool of the platform 1C:Enterprise. It should also be borne in mind that in any configuration, developers create objects that are also part of the type system.

Thus, the idea came up to create a tool for tracking type mismatch errors for the programming language of 1C:Enterprise. To achieve the goal, it was necessary to perform the following tasks:

1. Analyze the type system of 1C:Enterprise.
2. Develop a program that extracts information about types used in configurations of 1C:Enterprise.
3. Create an application for static type-checking in programs written in the programming language of 1C:Enterprise.

## 2. Principle of static type-checking

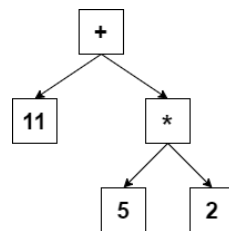
To solve problems in languages with dynamic type-checking, developers can use methods of static analysis of program code. The use of such tools in the development process allows you to identify areas where type mismatch errors occur before launching the program [1].

Static code analyzers check the correctness of using language constructs by examining the abstract syntax tree, and also determine method calls to simulate the behavior of objects at a certain point the execution of the program [2].

Abstract Syntax Tree (AST) – representation of the program code as a graph [3], in which:

- non-leaf vertices – expressions or composite instructions;
- leaves – operands or simple instructions.

For example, an arithmetic expression of the form  $11 + (5 * 2)$  will be displayed as an AST as shown in Figure 1.



**Figure 1:** Expression AST  $11 + (5 * 2)$

To build an AST, a lexical and syntactic analysis of the program is performed [4].

The purpose of lexical analysis is to identify lexemes in the program code, which include keywords, operation signs, some fixed values (literals), etc.

After defining the lexemes, a syntactic analysis is performed, the essence of which is to match the lexemes in accordance with the grammar defined for the programming language used. Based on the results of the parsing, you can build an AST.

### 3. Type system of 1C:Enterprise

The complexity of developing a static analysis tool for the programming language of 1C:Enterprise is caused by the complexity of the type system in 1C, which consists of platform types, as well as types of configuration objects that are created by configuration developers and belong to one of the metadata types [5]:

1. Common objects: Subsystem, Common Module, Role, etc.
2. Applied objects: Catalog, Document, Enum, Information register, Charts of characteristic types, etc.

Each type of metadata has a basic functionality that is defined in the platform 1C:Enterprise. When creating a configuration, developers create common and application objects that inherit all the functionality of the corresponding metadata types. This functionality can be redefined and supplemented to meet the needs of the business processes of the system being created.

It is important to note that some objects inherit functionality not from one, but from several types [6] that are defined in the platform 1C:Enterprise.

For example, an application object of the Catalogs metadata type inherits the functionality of the following platform types:

1. CatalogObject – provides the ability to read and change the elements of the Catalog.
2. CatalogRef – used to store a link to the Catalog item. It can be used to read the data of the Catalog item.
3. CatalogSelection – provided for traversing Catalog items.
4. CatalogList – intended for managing the list of Catalog items in the tabular field of the form.
5. CatalogManager – used to perform operations with the Catalog, for example, to search for an item by name or to add a new item.

All types of configuration objects have properties, the types of which can be both application configuration objects and data types of the built-in language of platform 1C:Enterprise. Information about such types is contained in the built-in 1C help, called Syntax Assistant, and is also available on the 1C:ITS portal [7].

Thus, for the analysis of the AST describing the configuration structure based on 1C:Enterprise, it is necessary to use information not only about the configuration objects, but also about the types of the built-in language.

### 4. Development of the Configuration Types Tree format

To develop a static type-checking tool, it is necessary to use information about configuration objects, as well as about the types provided by the platform 1C:Enterprise, to determine what types a property can have, as well as parameters or return values of methods. Therefore, it was necessary to generate a document describing all types of configuration and platform in order to apply this information when performing a static analysis.

## 4.1. Representation of the 1C:Enterprise configuration in XML format

Any configuration developed on the platform 1C:Enterprise can be uploaded to XML-format files in the form of a hierarchical structure in which all configuration objects are distributed in directories with a name corresponding to the common or application object.

After downloading the 1C:Enterprise configuration, the structure of the XML files of each configuration object differs due to their specifics. For example, for objects of the metadata types Catalog, Document, DataProcessors and Report, details and tables are defined, and all registers (information, accumulation, accounting and calculation) do not have tabular parts, and, in addition to the details, there are dimensions and resources.

For example, in the XML file for uploading an application object of the Catalog metadata type, the information is presented as follows:

```
<MetaDataObject xmlns="..." xmlns:xr="...">
  <Catalog>
    <InternalInfo>
      <xr:GeneratedType
        name="CatalogObject.Users"
        ↪ category="Object"
      />
      <xr:GeneratedType
        name="CatalogRef.Users" category="Ref"
      />
      ...
    </InternalInfo>
    <Properties>
      <Name>Users</Name>
      <StandardAttributes>
        <xr:StandardAttribute name="Ref"/>
        ...
      </StandardAttributes>
    </Properties>
    <ChildObjects>
      <Attribute>
        <Properties>
          <Name>BirthDate</Name>
          <Type>
            <v8:Type>xs:dateTime</v8:Type>
          </Type>
        </Properties>
      </Attribute>
      <Attribute>...<Attribute>
      ...
    </TabularSection>...</TabularSection>
    ...
  </Catalog>
</MetaDataObject>
```

```
        <Command> . . . </Command>
        . . .
    </ChildObjects>
</Catalog>
</MetaDataObject>
```

The elements denote the following:

- `InternalInfo` – contains an enumeration of the types that were created in 1C:Enterprise for this configuration object;
- `Properties` – defines the standard properties of the configuration object: `Name` (name) and `StandardAttributes` (standard details);
- `Attribute` – indicates the details of the configuration object, the acceptable types of the details are listed in the `Type` element;
- `TabularSection` – describes the data of the tabular part of the configuration object, which includes information about standard properties and details;
- `Command` – provides information about the configuration object command.

## 4.2. Complexity of the configuration representation format

Configuration upload XML files have a rather complex and redundant structure because they contain, in addition to the basic properties, behavior settings and display configuration data that do not carry information for the type matching task.

The complex structure of XML configuration upload is also due to the fact that the conversion works both ways: the configuration is not only uploaded to XML, but also completely restored from the uploaded files. Therefore, uploading a configuration to XML is a representation of objects without losing information, and only part of the data is enough to form a document that will contain a description of the types of configuration objects and the platform.

## 4.3. Format for representing platform types and configuration objects

Since the XML representation of the 1C:Enterprise configuration is redundant for using it as a type description when performing static analysis, the idea of presenting the configuration in a simplified format arose.

The simplified format for describing configuration objects and platform types in this work is called the "Configuration Type Tree" (CTT). This name is due to the fact that each configuration object inherits the functionality of one or more types defined in the platform 1C:Enterprise.

Thus, the CTT file should consist of:

- platform types (tree branches) – serve as the basis for configuration objects;
- configuration objects (tree leaves) – inherit the functionality of platform types.

The process of forming the CTT file consists of the following stages:

1. Defining the format and schema for storing data of configuration objects and platform types.
2. Implementation of configuration data parsing.
3. Creating a file consisting of data from configuration objects and platform types.

#### 4.4. Configuration Types Tree in XML format

The XML format is selected for storing types and methods [8]. This choice is due to the fact that for XML documents there are strict markup rules, due to which a single document structure is observed [9, 10]. It is also possible to use links between elements in XML, which was also an important criterion when choosing a format, since configuration objects can be linked together [11] and with platform types.

To represent the 1C:Enterprise configuration in CTT format, an XSD schema [12] has been created that describes the rules for forming an XML document. The description of this scheme in XML format is published in [13], and the structure is shown in Figure 2.

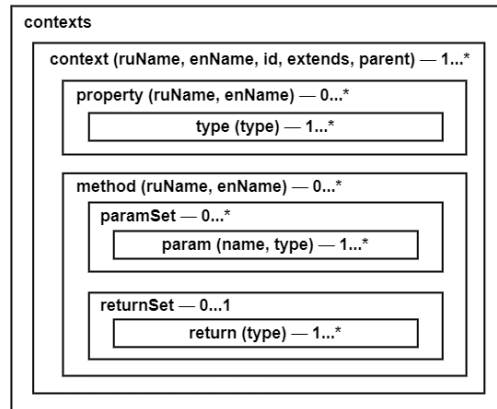


Figure 2: CTT in XML format

The elements denote the following data:

- `contexts` — root element of the XML document;
- `context` — description of a configuration object:
  - `ruName` and `enName` — name of the object in Russian and English;
  - `id` — identifier of the object that is used to specify references if the object is a type for a property;
  - `extends` — identifier of the `context` element from which the functionality is inherited;
  - `parent` — a reference to the element of which the current `context` is a component (specified only for table parts, forms, layouts, and commands of application objects).
- `method` — defining the method of object that can contain both mandatory and optional parameters;
- `paramSet` — sets of method parameter types, within which the parameters (`param`) are listed, where the attributes of each of them are the name (`name`) and the type (`type`);
- `returnSet` — set of possible types of the return value, that lists the return elements with the `type` attribute.

It is important to note that Russian and English names for some elements have been added due to the fact that in addition to configuration data, information about platform types, where objects, properties and methods have names in two languages, should also be converted into the CTT format. Information in English may be required when performing a static analysis, since it is permissible to use built-in 1C constructs in this language when English is selected as the interface language for working with the platform.

To automatically generate an XML description of the configuration in the CTT format, a program is written in the programming language of 1C:Enterprise. Parsing the source XML files with configuration data is performed using XPath queries, and the generation of an XML description in the CTT format is implemented using the XDTO tool.

Thus, all the types defined in the configuration are placed in CTT format file. A file of the same format was also created, in which all the types from the reference information of the platform 1C:Enterprise were added.

## 5. Static type-checking tool

To create a static type checking tool, it was necessary to get a representation of the configuration program code in the AST format, and then perform an analysis of the correspondence of types to language constructs, using information from the CTT format document.

Reading the code of 1C configuration modules was previously implemented in the programming language of C++ by the author of this work Alexander S. Balyuk. This task was performed to identify unused procedures and functions in an arbitrary configuration of 1C:Enterprises by constructing and analyzing a graph of method calls.

Thus, by the beginning of the research on the topic of this work, the foundation of the software package for the implementation of static analysis was already laid.

In order to search for type mismatch errors in 1C:Enterprise configurations, it was necessary to perform the following:

1. Implement the transformation of the method structure into an AST representation.
2. Design a static type-checking tool.
3. Organize the loading of the DTC format file into a program for static type-checking.
4. Write rules for calculating the types of each construct of the programming language of 1C:Enterprise.

### 5.1. Converting methods to an AST representation

In the developed program for identifying unused methods, which is the basis for implementing static analysis, the lexical analyzer generation tool – Flex (Fast Lexical Analyzer) [14] was used to determine language designs (lexical analysis), and GNU Bison [15] was used for syntactic analysis.

As part of this work, all the constructs of the programming language of 1C:Enterprise were defined and their transformation into AST fragments was implemented. For each configuration method, a separate AST is constructed in which constructs can have child vertices.

For example, the `return_instruction` construct may have a child vertex that denotes the return value:

```
virtual node* return_instruction(node* _nd = 0) {
    auto nd = new::tree::return_instruction;
    if (_nd)
        nd->childNodes.push_back(tree::node_ptr(_nd));
    return nd;
}
```

For example, the function construct, unlike `return_instruction`, has 3 child vertices that define parameters, variables, and an execution block for the method:

```
virtual node* function(
    int directive,
    bool proc,
    str_p identifier,
    node* nd_formal_param_list,
    node* nd_export_type,
    node* nd_var_block_list,
    node* nd_execution_block
) {
    auto nd = new tree::function(identifier, proc);
    nd->childNodes.push_back(tree::node_ptr(nd_formal_param_list));
    nd->childNodes.push_back(tree::node_ptr(nd_var_block_list));
    nd->childNodes.push_back(tree::node_ptr(nd_execution_block));
    return nd;
}
```

According to the same principle, the formation of vertices is performed for all constructions of the programming language of 1C:Enterprise.

## 5.2. AST analysis

To perform static type checking, it is necessary to define the types that the construction can accept, that is, a vertex or a leaf of an AST. Therefore, it is necessary to process each construction, taking into account that it can have nested data structures (child vertices).

First of all, an abstract `TreeNodeBase` class was created, which contains the basic implementation of methods for parsing AST fragments. For all constructs (method, parameter list, return value, conditional operator, etc.), when writing static type checking rules, classes are created that are inheritors of the `TreeNodeBase` class. In this case, all or only some methods can be overridden. This approach is due to the different order of parsing structures from each other.

The `childNodes` attribute in `TreeNodeBase` is a set of child vertices. The operations of the class have the following purpose:

- `str()` – describes the string representation of an AST vertex or leaf;



- `parse(...)` – checks that the types match the language constructs and determines the possible return types of the function;
- `types(...)` – defines the set of valid types of the expression corresponding to the AST subtree with the root at this vertex;
- `getContextItems(...)` – returns a set of properties and methods corresponding to the AST subtree with the root at this vertex.

Implementation of the basic functionality of methods in an abstract class:

```

virtual std::set<Context*> parse(Context* context, void* ext = 0) {
    std::set<Context*> result;
    for ( auto nd : childNodes )
        if ( nd ) {
            auto r = nd->parse(context, ext);
            result.insert(r.begin(), r.end());
        }
    return std::move(result);
}
virtual std::set<Context*> types(Context* context = 0) {
    std::cerr << "Node " << str()
        << " does not have any type." << std::endl;
    return { };
}
virtual std::set<Context::ContextItem*> getContextItems(
    const std::set<Context*>& contexts, bool only_visible = false
) {
    return { };
}

```

Also, in the developing tool for static type-checking of program code, the load of the CTT format file received as part of this work was implemented. Thus, the sequence of operations performed in the program for static type-checking is shown in Figure 3.

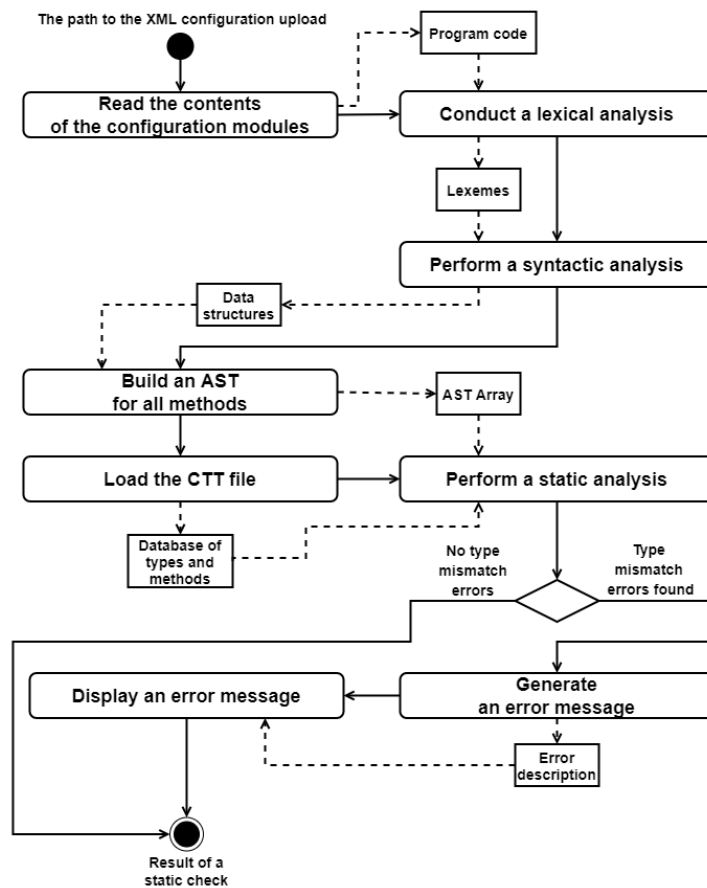
### 5.3. Testing

The functionality of the developed static type-checking tool was checked on the "1C:Standard Subsystem Library" configuration. At the time of testing, version 3.1.3 was current.

As a result of test runs of the program the following errors were detected in the configuration "1C:Standard Subsystem Library":

- incorrect parameter passing to a method;
- access to a non-existent property of an object;
- treating a variable of simple type as a collection.

For example, a value of the Number type will be passed to the method of the built-in `StrReplace(<String>, <String>, <String>)` language containing three parameters of the String type, then the program will output the following message: incorrect parameters were passed to the `StrReplace()` method.



**Figure 3:** Program for static type-checking

## 5.4. Project development

According to the project development plans, firstly, it is planned to implement a search for errors in the use of user interface elements, since at the moment the structure of elements on user forms is not processed.

As part of the project, to perform static type checking of 1C:Enterprise configurations, which currently checks the correspondence of types to language constructs, in the future it is planned to implement a static analysis of the 1C:Enterprise query language, which is used to obtain information from the database.

## 6. Conclusion

As a result of the work, a mechanism was created that allows you to find type mismatch errors in programs written in the programming language of 1C:Enterprise.

The developed software package can become a necessary tool in the process of creating

configurations on the platform 1C:Enterprise, since it will find a significant number of errors in programs before transmitting them to users.

It is also important that the tools for static code analysis of 1C:Enterprise configuration modules can be further developed and improved.

## References

- [1] M. Y. Tokarenko, F. F. Bukanov, Methods of automated static analysis of program code, in: Actual problems of information security. Theory and practice of using software and hardware: Materials of the X All-Russian Scientific and Technical Conference, Samara, March 21-22, 2017, Samara State Technical University, Samara, 2017, pp. 138–142.
- [2] M. K. Ermakov, S. P. Vartanov, Dynamic analysis of arm elf shared libraries using static binary instrumentation, Proceedings of the Institute of System Programming of the Russian Academy of Sciences 27 (2015) 5–24.
- [3] J. Jones, Abstract syntax tree implementation idioms, Pattern Languages of Program Design (2003). URL: <https://www.bibsonomy.org/bibtex/2a86095c92a9801ab24d2196d334abe46/gwpl>.
- [4] M. A. Kuznetsov, A. V. Khorolskiy, Theoretical aspects of developing of compiler for learning the basics of translation, Modern problems of science and education (2013) 52–59.
- [5] 1C:Enterprise configuration objects, 2021. URL: <https://its.1c.eu/db/metod8dev/content/2579/hdoc>.
- [6] Working with application objects using the built-in language, 2021. URL: <https://its.1c.ru/db/metod8dev/content/2698/hdoc>.
- [7] Information and technological support of 1C, 2021. URL: <https://its.1c.ru/>.
- [8] Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation 26 November 2008, 2008. URL: <https://www.w3.org/TR/xml/>.
- [9] XML Data Management: Approaches to defining XML documents, 2019. URL: <http://citforum.ru/internet/xml/harold/>.
- [10] J. Tekli, R. Chbeir, K. Yetongnon, An overview on xml similarity: Background, current trends and future directions, Computer Science Review 3 (2009) 151–173.
- [11] A. S. Balyuk, V. A. Popova, Static type-checking for programs developed on the platform 1c:enterprise, Bulletin of the Irkutsk University (2020) 54–55.
- [12] W3C XML Schema Definition Language 1.1 Part 1: Structures. W3C Recommendation 5 April 2012, 2012. URL: <https://www.w3.org/TR/xmlschema11-1/>.
- [13] V. A. Popova, A software package for converting the 1c:enterprise configuration into a uml model, 2021. URL: <https://gitlab.com/converting-1c-to-uml/software-package>.
- [14] Lexical Analysis With Flex, 2020. URL: <https://westes.github.io/flex/manual/>.
- [15] GNU Bison — The Yacc-compatible Parser Generator, 2020. URL: <https://www.gnu.org/software/bison/manual/>.