### Information and Data Reporting

We can classify the data presented on screen to the user during the Scior execution in four different categories: user interaction, information, incompleteness, and inconsistencies.

#### User Interaction

User interaction is only required in for the five general rules and is only going to be required if the user runs Scior in the interactive mode (i.e., if the user does not set the software as automatic, as being interactive is the default behavior of Scior). The user is required to interact every time there is more than one option to fulfill a gUFO constraint.

The interactions required by Scior are simple, basically comprising selecting provided options. In most interactions, the user can (i) inform if a class should be classified or (ii) select a class among a list of classes to be classified into different gUFO categories, according to the rule being executed. Finally, the user can skip the selection—we provide this option for the case where the user does not know the required information. In all mentioned possibilities, Scior provides the user with basic information about the situation identified (gUFO constraint or model incompleteness) and about the involved classes.

#### Reporting Information

Besides the user interactions, other information presented on screen during the Scior’s execution are related to important execution steps (e.g., “*Starting GUFO types’ hierarchy rules ...*”), decisions took by the algorithm in certain rules (e.g., “*The class X is the unique possible identity provider for Y. Hence, it was automatically asserted as gufo:Kind*”), and confirmations. By the end of the execution, Scior presents [final statistics](#execution-statistics). In addition, the user can chose to print execution times through setting a [specific argument](https://github.com/unibz-core/Scior/blob/main/documentation/Scior-Execution-Modes.md#complementary-arguments) .

#### Reporting Incompleteness

Some important pieces of information presented on screen are the incompleteness cases found during the software execution. As [incompleteness is an ontological deficiency](https://research.utwente.nl/en/publications/ontological-foundations-for-structural-conceptual-models) , the automatic identification of incomplete classes is a valuable feature provided by Scior. During the execution, the software reports the incompleteness cases as warning messages and, after its end, it presents a table with incomplete classes and their respective detection rules.

Six of the Scior implemented rules can detect and report incompleteness. We present in the list below these rules and the situations that they can detect incompleteness:

* n\_r\_t: Identifies when there is no identity principle associated with a given class.
* ns\_s\_spe: Identifies when a class is associated with less than two gufo:Kinds.
* nk\_k\_sup: Identifies when a gufo:Sortal class does not have an identity provider (a direct or indirect superclass of type gufo:Kind).
* s\_nsup\_k: Identifies when a class does not have an identity provider.
* nrs\_ns\_r (applied only to complete models): This rule identifies when a class is gufo:NonRigid and gufo:Sortal without siblings, stating that the class must be set as a gufo:Role. It also informs that if the class is a gufo:Phase, at least another gufo:Phase sibling class is missing, representing an incompleteness.
* ks\_sf\_in: As phases always occur in phase partitions, this rule identifies when a class is the only gufo:Phase subclass of another given class.

Note that the ontology modeler must not always solve all cases of incompleteness, as some models may present incompleteness because of the level of abstraction adopted by their modelers. Also, incompleteness only occurs in models that are set as incomplete by the user. If the user states that the model is complete using the [specific argument](https://github.com/unibz-core/Scior/blob/main/documentation/Scior-Execution-Modes.md#models-completeness-modes) for that, then the mentioned problems are not incompleteness, but **inconsistencies**.

#### Reporting Inconsistencies

Differently from incompleteness, inconsistencies are not a type of ontological deficiency, but an invalid state that invalidates the execution of the software. In simple terms, it is an error that the ontology has and, hence, Scior reports these cases with error messages and aborts its execution so the ontology modeler can solve the problem.

In Scior, an inconsistency is detected whenever a type has to be set and the software does not find it as a valid option. I.e., Scior tries to move the type from the can\_type to the is\_type or to the not\_type list, but the type is not available in the can\_type list. Unfortunately, the current version of the software does not clearly specify the problem that caused the inconsistency in most cases—however, this is [a known issue](https://github.com/unibz-core/Scior/issues/7) that is going to be addressed soon.

The first example of inconsistencies that we can mention are the ones we presented in the past section, the ones that happen when a user sets an incomplete model as complete. The other cases are mainly related to incorrect information inputted to the software (e.g., the input ontology has a class of type gufo:Kind with a subclass of the same type). Finally, note that the information provided by the user during the interaction rules can also lead to invalid states, generating inconsistencies. We tried to avoid this last situation as much as possible with the creation of restriction rules. However, because of the iterative coding, it may happen.

## Output Files

This section presents information about the output files generated by Scior, which are the ontology added with the discovered ontological classifications, and the execution report file. Scior also creates a log file in a folder called logs inside the project’s folder, but the description of this file is out of this document’s scope.

### Output Ontology

After the Scior execution, all knowledge discovered through the execution of gUFO rules are added to the original graph ( i.e., to the input ontology graph) and this complete graph is saved as a *ttl* file. The file is saved into the same folder as the input file using the following file nomenclature: example-{YYYY.MM.DD-hh.mm.ss}.out.ttl, where “example” is the name of the ontology and the curly brackets are substituted by year (Y), month (M), day (D), hour (h), minutes ( m), and seconds (s) of the end of the execution.

By default, Scior registers only the new statements that make the ontology classes instances of gUFO types (in the format ontology\_class rdf:type gufo\_class) in the output file. It is also possible to import gUFO or even save all its statements in the output file [by using specific arguments](https://github.com/unibz-core/Scior/blob/main/documentation/Scior-Execution-Modes.md#complementary-arguments) .

### Report File

Scior generates a complete report file after concluding an execution. The report contains information about the execution itself and about the situation before and after its conclusion. The information presented in the report is:

* **Execution Information:** displays general and specific (per rule) execution times, together with computer and software specifications, used configurations, generated files and solution’s hashes.
  + The software creates solutions’ hashes so the user can quickly verify if the results have changed from one execution to another.
* **Lists of Classes Before Scior:** displays the classes grouped according to the amount of gUFO knowledge known \*\* before\*\* the Scior execution. I.e., it presents the **initial** state of the ontology that is going to be evaluated.
  + The categories used for grouping the classes are: Totally Unknown Classes, Partially Known Classes, and Totally Known Classes.
* **Lists of Classes After Scior:** displays the classes grouped according to the amount of gUFO knowledge known \*\* after\*\* the Scior execution. I.e., it presents the **final** state of the ontology that was evaluated.
  + The categories used for grouping the classes are: Totally Unknown Classes, Partially Known Classes, and Totally Known Classes.
* **Results` Statistics:** presents tables with total numbers and percentages of classes and classifications before the execution, after the execution, and the difference between these two situations.
  + The [next subsection](#execution-statistics) details the information about the presented statistics.
* **Incomplete Classes Identified:** presents a table containing all classes identified as incomplete together with the implemented rules that were responsible for that identification.
* **Knowledge Matrix:** As there are 14 gUFO Endurant Types, the knowledge matrix is a 15x15 matrix. Each matrix element indicates a QUANTITY of classes: the rowsindex (from 0 to 14) indicates how many known types \*\*before\*\* the execution, and the columns index (from 0 to 14) Shows how many known types **after** the execution. The position (ROW, COL) indicates how many classes began with ROW known types and ended with COL known types.
  + As instance, if matrix position (0,5) stores the value 17, it means that 17 classes started the evaluation (i.e., the user provided them as input) without known classifications and these classes finished (i.e., Scior provided them as output) with 5 known gUFO types.
* **Final Classes’ Classifications:** presents the complete internal gUFO classifications’ lists (is\_type, can\_type, and not\_type) for each class, sorted by their URI. With this information, the user can know the final classification of each one of the ontology classes.

The user can access the generated report, saved in [Markdown](https://www.markdownguide.org/basic-syntax/) format, at the reports folder inside Scior’s project folder.

## Execution Statistics

Scior presents to the user statistics about its current execution in two different forms: on screen and in the report file. The statistics are about data measured **before** and **after** the software’s execution and about the \*\* difference\*\* between these values. Their presentation contains the value of the item being measured and this value’s percentage with relation to its total amount.

You can find below an example of the statistics that are presented after the Scior’s execution:

Results of Scior execution when evaluating 34 CLASSES considering only TYPES:  
| Evaluation | Before | After | Difference |  
|------------------------:|---------------:|---------------:|---------------:|  
| Totally unknown classes | 30 (88.24%) | 12 (35.29%) | -18 (-52.94%) |  
| Partially known classes | 0 (0.0%) | 6 (17.65%) | 6 (17.65%) |  
| Totally known classes | 4 (11.76%) | 16 (47.06%) | 12 (35.29%) |  
  
Results of Scior execution when evaluating 476 CLASSIFICATIONS considering only TYPES:  
| Evaluation | Before | After | Difference |  
|------------------------:|---------------:|---------------:|---------------:|  
| Unknown classifications | 420 (88.24%) | 206 (43.28%) | -214 (-44.96%) |  
| Known classifications | 56 (11.76%) | 270 (56.72%) | 214 (44.96%) |

As you can see above, the presentation displays data about two items: the number of **classes** and **classifications**.

The first item presented is the number of classes that were evaluated. Scior presents it according to three classifications, which are the number of:

* Totally Unknown Classes: classes that do not have relations to any gUFO concept.
* Partially Known Classes: classes that have relation to higher-level gUFO concepts, but that are not yet fully classified.
* Totally Known Classes: classes that have relation to higher-level gUFO classifications. I.e., classes that have their classifications totally known.

Regarding classifications, these are the gUFO classes (e.g., gufo:Kind, gufo:Sortal, gufo:RigidType) to which an OWL ontology concept can be mapped (for endurant types, this is done via an rdf:type predicate). Knowledge about classifications: the number of known and of unknown classifications, as well as the total number, which is the sum of the first two ones.