# Data Exchange in Bauhaus

Jörg Czeranski, Thomas Eisenbarth,
Holger M. Kienle, Rainer Koschke,
Erhard Plödereder, Daniel Simon,
and Yan Zhang
University of Stuttgart, Germany
bauhaus@informatik.uni-stuttgart.de

Jean-François Girard and Martin Würthner
Fraunhofer Institut für Experimentelles
Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
{girard, wuerthne}@iese.fhg.de

## Abstract

*In the context of the Bauhaus project, reengineering environments to support program understanding of legacy code are being developed. Bauhaus defines two formats to represent information that has been extracted from source code. One of these formats, RG, is suitable as an exchange format. This paper introduces RG, describes how it is represented as an exchange format, and discusses schema conversions in RG.*

## 1. Introduction

The Bauhaus project performs research on techniques to support program understanding of legacy code and more specifically on the recovery of the system's architecture, which consists of its components, connectors, and constraints. Information about the system is exclusively extracted from the source code (often this is the only reliable source of information) in a semi-automatic way that actively involves the user of one of these environments.

To model source language information, the Bauhaus tools use two representations:

- The *InterMediate Language* (IML) [1] models source language information precisely (i.e., the semantics of the programming language constructs is preserved), which makes it possible to run sophisticated data-flow and control-flow analyses (e.g., slicing and trace recovery) on top of it. IML is essentially a persistent attributed tree representation as it is typically employed in compilers to perform in-depth analyses of dynamic semantics.

- The *Resource Graph* (RG) abstracts from a particular source language by only representing global information such as call, type, and use relations. This representation is used by analyses that do not need the fine-grained representation of the source code that IML provides. For example, component recovery operates on the RG. Its result consist of hierarchical groups of RG nodes.

Thus, IML is used to model the *source code level*, whereas RG models the *logical/design level*. In principle, RG could be used to model the source code level as well, but its representation would be very inefficient in space and time compared to IML.

Typically, a Bauhaus front-end translates from the source language to IML. A separate pass translates from IML to RG. (Alternatively, a front-end can directly generate RG, but this means that IML analyses cannot be run.) IML is not designed to be a flexible exchange format, nor does it incorporate concepts to represent all levels of abstraction (e.g., logical/design or architecture/conceptual level). In the rest of this paper, we focus on RG.

The Bauhaus project is a cooperation between the Bauhaus Group at the University of Stuttgart and Fraunhofer IESE. In the beginning of the project, the two partners shared a common code base implemented in Refine. Since then, more loosely coupled development at both sites has taken place resulting in two distinct, yet related system implementations, where each system has its own extraction tools. Within our collaboration, the RG format is used to exchange data between the two systems.

## 2. Resource Graph

Bauhaus uses RG to keep structural information that is directly obtained from source code. This information can be described by an entity relationship model. The entities are programming language constructs (e.g., functions, types, and variables) and abstract concepts, which typically have been extracted by certain analyses (e.g., abstract data types, components, and subsystems). Examples of relationships range from information that can be directly extracted from the source code (e.g., function calls) to more abstract concepts (e.g., communication between a client and a server). Entities are represented as nodes and relationships between the entities are represented as edges. The resulting graph then can be visualized and manipulated with a graph editor. (We use a modified version of the Rigi editor [3].)

### 2.1. Schema

This section describes RG's base entities and relationships. Both are structured in separate type hierarchies (see Figure 1). In addition to the base entities and relationships, several extensions have been defined to model information of specific analyses (e.g., atomic component detection). These extensions are naturally expressed via subtyping from existing entities and relations.

*base entity*
- *architectural quark*
  - subprogram
  - *object*
    - variable
    - constant
  - user-defined type
- *record components*
  - record comp. specifier
  - record comp. instance

*base relationship*
- call
- actual-parameter-of
- *signature-type*
  - parameter-of
  - return
- *reference*
  - take-address-of
    - obj-address-of
    - comp-address-of
  - set
    - obj-set
    - comp-set
  - use
    - obj-use
    - comp-use
- enclosing
- part-type
- of-type
- same-expression
- local-obj-of-type

**Figure 1. Type hierarchy of base entities (left) and relationships (right).**

The entities derived from *architectural quark* (i.e., user-defined types, global subprograms, variables, and constants) are the smallest significant elements at the architectural level. In addition, record components in a type declaration and record components of instances of the type are modeled as two separate entities derived from the common base entity *record component*. To summarize, the base entities can model the constructs usually found in procedural programming languages that have a bearing on architecture recovery.

The relationships between entities should be fairly self-explanatory. For example, the *signature-type* of a subprogram is a type that occurs in its signature either as return type (*return*) or parameter type (*parameter-of*) and an object can be referenced by using its value (*use*) or taking its address (*take-address-of*); a variable can additionally be set (*set*). A type $T_1$ can be used in the declaration of another type $T_2$. In this case, we consider $T_1$ a *part-type* of $T_2$. ($T_2$ is the composite type of $T_1$). The resulting entity-relationship model is depicted in Figure 2.

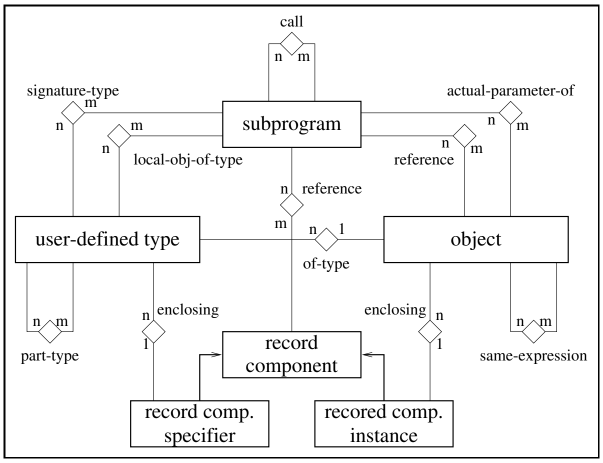


**Figure 2. Schema of base entities and relationships.**

### 2.2. Exchange Format

In order to make RG data persistent, a specific exchange format has been designed. Since this format is used to exchange RG data between two independent research groups, its objectives are almost identical with the ones that have already been identified for a general exchange format [1, 2]. Our format has the following characteristics: It is (1) human-readable, but (2) still relatively compact, (3) can be expressed with a simple and small grammar, and (4) is extensible.

A human readable format simplifies debugging of the generated output and allows a user to understand and modify files without fancy tool support. Despite this feature, the files are relatively compact because the format reduces space by avoiding redundancy. For example, a writer can utilize string references to share identical strings. A simple grammar without nested structures allows us to use a scanner instead of complete parser. This is important in our heterogeneous setting where readers and writers are written and maintained in multiple languages (Ada, Refine, and C++). The format is extensible in the sense that arbitrary attributes can be attached to entities and relations.

Our format shares many characteristics with graph formats. Entities correspond to nodes and relations to edges in a graph. Every node/edge has a unique id that can be subsequently used to refer to it. The resulting graph is typed because every specified node/edge must be given an explicit type name. Since types names can be long and appear frequently, abbreviations are introduced for them (in the NODE_TYPES and EDGE_TYPES sections). For example, the graph

$$\mathtt{i}_{variable} \longrightarrow_{of-type} \mathtt{integer}_{type}$$

is represented as follows:

```
NODE_TYPES
> variable NT1;
> type NT2;
EDGE_TYPES
> of-type ET1;
NODES
#1 i NT1 list-of-attributes ;
#2 integer NT2 list-of-attributes ;
EDGES
#1 #2 ET1 list-of-attributes ;
```

Our RG supports multiple *views* of a system [1]. Views are consistent subgraphs that are used to represent different user perspectives as well as results of certain analyses in a single graph. Typically, a node or an edge is present in different views. For example, a call-graph view can restrict the graph to contain only *subprogram* entities and *call* relations.

Every file that adheres to our format starts with a standardized header that contains the version number of the exchange format, the name of the schema that defines the semantics of the nodes and edges, the version of the schema, and the name of the producer. The schema plays an important role for the reader because it allows the conversion of certain node/edge types or the transformation of the graph structure according to semantic differences of the model it imports from (see Section 2.3 for further explanation). Following the header, the file is divided in (optional) sections that specify views, attributes, strings, type mappings, nodes, and edges. Entities that have been introduced in the respective sections can be shared later on via references.

### 2.3. Schema Conversions

Differences in underlying schema occurred because the two Bauhaus partners evolved different toolsets. These differences occur at three levels:

- the abstract information contained in the data (e.g., call graph or data-flow information)
- the schema of the data, i.e., the structure in which the information is represented (e.g., the same information may be expressed as an additional attribute or a new relation)
- the raw data

The differences are only relevant if the emitted information is of interest to the user of the data, e.g., if we are only interested in the call graph, the information on types and the schema to describe this information can be ignored by the user. If the information and the schema is the same, the data can be exchanged without any transformations; in all other cases, the data need to be converted. If the information is the same but the schema differs, the following conversions are necessary:

1. If the schemas are isomorphic, i.e., they differ only in names of nodes, edges, or attributes (e.g., many of our relations are slightly differently named in the two systems), schema conversion is performed by simple renaming.
2. If the schemas differ in nodes, edges, or attributes (e.g., one specific information is expressed in one system as an attribute of a relation and in the other system with an explicit relation), schema conversion needs to be done by semantic-preserving graph transformations.

If the information differs completely, a schema conversion is not possible, but then the data would not be exchanged anyway, i.e., in practice, we may assume that provided and expected information at least overlap. If the provided information is a partial subset of the needed information, the reader gets incomplete information. If the provided information is richer than needed, part of it can be ignored.

Extraction tools are another important issue. Even though the abstract information and the schema may be the same, the exchanged information may still differ in the transferred actual data. For example, if function pointers are resolved, different static analyses with varying precision may be used. In this case, schema conversion will not help.

Ideally, we reach a consensus on the exchanged information and the schema to represent it. However, in practice such an agreement will be hard to achieve because of the investment spent in tools around an evolved schema—just to mention one reason. Hence, beyond the definition of the exchange format, tools should be made available that support schema conversions.

## References


[1] R. Koschke, J.-F. Girard, and M. Würthner. An intermediate representation for integrating reverse engineering analyses. *Proceedings of 5th Working Conference on Reverse Engineering (WCRE '98)*, pages 241–250, October 1998.

[2] S. E. Sim, R. Holt, and R. Koschke. Workshop on standard exchange format at ICSE 2000. http://www.cs.toronto.edu/~simsuz/wosef, June 2000.

[3] S. R. Tilley, H. A. Müller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. *Proceedings of Conference on Software Maintenance (CSM '94)*, pages 142–151, September 1993.