Where’s The Schema?
A Taxonomy of Patterns For Software Exchange

D. Jin, J.R. Cordy and T.R. Dean
Department of Computing and Information Science
Queen’s University, Kingston, Canada

Abstract
Reverse Engineering and Program Comprehension tools extract, organize and analyze information about the design and implementation of software systems. Before tools can exchange information, they must share, at some level, the organization for the data exchanged. That is, they must share a schema. In this paper we examine the various ways in which schemas are represented and used in tools. Schema use is classified according to how and where a schema is defined, leading to the identification of four patterns of exchange. We examine these exchange patterns and discuss how each has been used in existing tool integration technologies. An evaluation of each exchange pattern against the requirements for a standard exchange format reveal the pattern of schema use that is most suitable for integrating tools.

1 Introduction
Reverse Engineering and Program Comprehension tools extract, organize and analyze information about the design and implementation of existing software systems. Efforts towards integrating reverse engineering tools have been severely hindered by the lack of a consistent model for the structural makeup of software representations. The development of a Standard Exchange Format (SEF) for software is seen as the desired solution to this problem [31, 24, 12].

Schemas that accommodate the representational needs of various reverse engineering tools are an essential part of an SEF. In this paper, we examine the various ways in which schemas are represented and used in tools, and classify them according to two distinguishing characteristics. Based on the use of schemas, four different exchange patterns are distinguishable. We examine each of these exchange patterns and discuss how each has been used in existing reverse engineering tool integration technologies. We also examine how each of the exchange patterns satisfy the thirteen requirements given by St-Denis et al. [32]. This leads us to propose which of the exchange patterns is best suited for use in software exchange formats.

2. Integration Technologies
In this paper we restrict our discussion of integration technologies to those that have been put to use for representing software in the context of reverse engineering. In particular we refer to the following:


Annotated Terms (ATerms) [36, 35] An exchange format and an API that represents data produced by parsers, structural editors, compilers and other components in software reengineering tools.

InterMediate Language (IML) [7] A portable intermediate representation developed by the Bauhaus Project [2].

Resource Graph (RG) [7] An exchange format for medium to high abstractions of source code developed by the Bauhaus Project.

Common Object-based Re-engineering Unified Model (CORUM) [38] An API-based environment for integrating software reengineering tools that work at the source code level.

CORUM II [19] A proposal for enhancing CORUM to provide advanced functionality for analysis at the architectural level of abstraction.

Datrix-TA [21] A format for exchanging Datrix-formatted abstract semantic graphs (ASGs) [20] among the different tools that make up the Datrix system.

FAMOOS Information Exchange Model (FAMIX) [33, 8] A portable intermediate representation for object-oriented source code.
Graph eXchange format (GraX) [9, 10] A format for exchanging software representations as TGraphs [11].

Graph Exchange Language (GXL) [15, 16] A flexible format for exchanging software representations at all levels of abstraction.

PROgramming with Graph Rewriting Systems (PROGRES) [29, 30] The format used in the PROGRES environment, an integrated set of freeware tools that help developers create, analyze, compile and debug specifications for graph rewriting systems.

Rigi Standard Format (RSF) A format for exchanging high-level representations of software systems used by the Rigi [26] tool.


TA++ [22] A modified version of TA used for representing and manipulating software representations among components that make up the TkSee [34] tool.

These integration technologies can be distinguished using characteristics such as [18]:

- The data structure used to represent software.
- The level of abstraction supported.
- The encoding method used.
- The mechanism used to transfer representations among reverse engineering tools.
- The ability to change the structure and interpretation of data represented.

All of these integration technologies share a common purpose: to enable the portability of structured information among different systems. Within the reverse engineering domain, representations of software are the structured information whose exchange is enabled. Almost all representations are constructed using some variant of an entity-relationship (E-R) model [5]. That is, E-R models provide a clean separation between the information that defines the allowable characteristics of a model and the data that is represented in the model. The former is known as schema and the latter as instance.

In terms of integration, this schema-instance severance provides a significant advantage. Negotiation of exchange among tools requires communal knowledge of the structure of information to be passed. For a given software model, this information is readily available in the schema. So the schema plays a pivotal role in the exchange process.

3 Schema Classification

Ultimately, the way a schema is used dictates how a tool will negotiate exchange with other tools. We classify the use of schemas into two categories: schema definition and schema locality.

3.1 Schema Definition

The Schema Definition category characterizes how the schema is defined. Within this category two classifications are identified:

Implicit. The structure of the representation is implied by the context in which the representation is used.

Explicit. The structure of the representation is provided, either through a specification or some other means.

3.2 Schema Locality

The Schema Locality category distinguishes where the schema is defined. Within this category two classifications are identified:

Internal. The schema is an integral part of the tool. As a consequence, the schema is not required to participate in the exchange.

External. The source for the schema definition is external to the tool. Because of this detachment, the schema is a participant in the exchange that occurs between tools. The schema is received by each of the tools either simultaneously with instance data or separately as a precursor to subsequent transmissions of instance data.

4 Exchange Patterns

According to the schema classifications outlined above, four different types of exchange can be negotiated among reverse engineering tools. We refer to these types of exchange as exchange patterns. We will now characterize the exchange patterns and provide examples for each. Figures 1 to 5 show the relationship between the tool, the schema and the representation of the software. Each figure consists of the following components:

- A tool T.
- A schema S.
- I and I’ respectively representing the state of a software representation instance before and after it is processed by tool T.
In Figure 1 an exchange that uses an implicit/internal schema is shown. The schema is embedded in the code, so it is found in many locations within the tool. Tools that are built to make use of an API to exchange software representations fall into this category. APIs essentially have a fixed schema, so the tools that use them are constructed according to an implicit yet predetermined concept of the software representation being exchanged.

Exchange that uses an explicit/internal schema is shown in Figure 2. Although the schema remains an integral part of the tool, it is provided as a specification so the schema is shown in a single location. Tools constructed within the PROGRES environment make use of schemas in this fashion. The tool developer first provides a schema in the form of a specification that outlines the graph-based structure of data to be represented and the operations that can be performed on them. Transactions that work with graph instances provide the functionality for the tool being constructed.

In Figure 3 and Figure 4 we see two exchange varieties that use an explicit/external schema. The tool shown in Figure 3 receives the schema first followed by the instance data. The integration technologies GraX and GXL work in this fashion. Schema and instance data are stored separately and all data instances provide a link to the file where the schema is stored. In Figure 4 the schema and the instance data are received simultaneously. The integration technology TA works this way. The schema information stored in the scheme tuple and scheme attribute sections are exchanged along with instance data stored in the fact tuple and fact attribute sections of the same file.

Exchange that uses an implicit/external schema is shown in Figure 5. In this case, the schema does not exist (so it is shown in a box with a dashed border) yet it does dictate the structural semantics of the information exchanged. The integration technology RSF is an example of an exchange format that works this way. The use of a tuple notation is a syntactic requirement. The implicit schema for the information exchanged is an unconstrained E-R model. Tools such as Rigi [26] and Holt’s Grok [14] accept E-R models in RSF. These tools have been pre-configured to handle constraint-free E-R instance data, so there is no need for a schema.

The exchange patterns used for each of the reverse engineering tool integration technologies are shown in Table 1.

5 Advantages and Disadvantages

As we mentioned, schema definition is a characterization of how the schema is defined, while schema locality relates to where the schema definition takes place. We now consider the advantages and disadvantages of each schema classification category in relation to their use in exchange.
5.1 Implicit Schema Definitions

The main advantage of an implicit schema definition is that it provides good performance. There is no need to carry out any additional processing or manage specifications for the representation being used. When the implicit definition is located internally, the representation is close at hand, being built into the code for the tool. Even when the implicit definition is external, the tool knows the structure of the information being exchanged so the opportunity to handle it appropriately is provided. This typically translates into the ability to process large quantities of information in a fast and efficient manner. This is especially beneficial for reverse engineering tools that work with source whose magnitude is measured in millions of lines of code.

A number of disadvantages offset the performance advantages of an implicit schema definition. Because the definition is static by nature, the representation is not extensible. This is a major problem for tools that are built around a particular information model. In such a situation, making changes to the representation involves a wholesale revision of code. Documentation is also a problem when the schema definition is implicit. A separate document outlining the structure and semantics of the representation is a necessity. Maintaining this documentation is time consuming and keeping it in sync with tool or exchange format changes is especially challenging.

A third problem with implicit schema definitions relates to the manner in which tools typically accept input. It is often useful to verify the integrity of information being exchanged. This usually involves a check to ensure that the input is well-formed. When the schema definition is implicit, such a test is difficult to implement and maintain. The tool functions that handle software representations are often deeply embedded and widely distributed throughout the code for the tool. A test that effectively checks incoming information must be based on all uses of the representation by the tool. In addition, the check must stay in sync with any modifications that are made over time to the exchange format or the internal representation within the tool. The effort involved in creating such a check in essence duplicates the efforts originally involved in handling the representation within the tool in the first place. As a consequence, it is unlikely that a tool that negotiates exchange using an implicit schema definition will include a check for well-formed input.

5.2 Explicit Schema Definitions

Exchange involving an explicit schema definition offers many benefits. The tool makes use of a specification or some other explicit means that identifies the structure and semantics of the information input. A clear separation exists between schema and instance data, no matter if the schema definition is internal or external. Because the definition is dynamic by nature, the representation is highly extensible. Modification of the representation is easily accomplished through changes made to the schema specification. All the information relating to the representation is located in a single location. This makes it easier for humans to get an overall understanding of the structure and semantics supported. The representation is always well documented and up to date. The explicit definition for the schema is itself the documentation.

Checking for well-formed input is a straightforward process when the schema definition is explicit. The schema specification holds all the requirements that must be satisfied for the information to pass the checker. Implementing the checker is simple because the schema specification is complete and close at hand. The checker does not need to be maintained because the schema specification always outlines the current representation in use. For these reasons, it is likely that a tool that negotiates exchange using an explicit schema definition will include a check for well-formed input.

The main drawback of an explicit schema definition is that it requires interpretation. The schema must be processed first before the tool can accept instance data. This intermediate step ultimately affects the performance of the tool. More importantly, there is a requirement for the tool to orient itself towards the representation provided in the schema. The tool must be flexible to accommodate this kind of functionality. In a best-case scenario, the tool would be able to accommodate any representation. In reality, it is

<table>
<thead>
<tr>
<th>Schema Localit</th>
<th>Implicit</th>
<th>Explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>ASIS, CORUM, CORUM II, IML</td>
<td>ATerm, PROGRES</td>
</tr>
<tr>
<td>External</td>
<td>RSF</td>
<td>Datrix-TA(^1), FAMIX(^2), GraX(^2), GXL(^2), RG(^1), TA(^1), TA++(^1)</td>
</tr>
</tbody>
</table>

Table 1. The Exchange Pattern Used By Various Reverse Engineering Tool Integration Technologies

\(^1\)Employs simultaneous receipt of schema and instance
\(^2\)Employs consecutive receipt of schema and instances(s)
likely that the representational capabilities of many tools will be limited. Building flexibility into a tool may also add significant complexity to the development effort.

5.3 Internal Schemas

The main advantage of an internal schema is accessibility. The tool does not need to venture out to an external source to determine the structure and semantics of the information model. This is an obvious advantage in terms of performance.

The difficulty with an internal schema definition becomes apparent when there is a need to change the information model. Maintaining conformity among two or more tools is difficult to achieve. This is especially challenging when the schema definition is implicit in all the affected tools. Changes must be implemented exhaustively throughout the code for each of the tools affected. Clearly an internal schema tends to make all tools participating in the exchange conform to a rigid representational structure and semantics.

5.4 External Schemas

With an external schema definition, managing conformity among two or more tools participating in the exchange is easily accomplished. A single schema definition is all that is necessary to ensure that each tool is using the correct structure and semantics for the representation being exchanged. Complete representational conformity among each tool participating in the exchange is assured as long as each tool makes use of the same schema definition. An external schema definition eliminates the need for a complete overhaul of the code for each tool when a change is made to the representation.

Nevertheless, the rules that each tool uses to process and analyze exchanged information can come out of sync with the schema because its definition is separated from the tool. Maintaining consistency between the code for a tool and the schema is challenging. The problem is exacerbated by the fact that external schemas are easily changed. The more often a schema is changed, the more likely that a loss of consistency will occur. The code in essence defines what the tool does with the information once it is successfully exchanged. But how this is accomplished is completely dependent on the structure and semantics of the representation defined externally by the schema.

5.5 Comparative Summary

Table 2 summarizes the relative advantages of disadvantages of each of the schema classification categories mentioned in this paper.
Advantages | Disadvantages
--- | ---
**Implicit** | • High performance no matter how large the input | • Not extensible
• Highly Extensible | • Low Performance
• Easily Understood | • Tool code is more complicated
• Well documented | • Difficulties managing changes among two or more tools
• Check for well formed input is easier to implement | • Keeping the code consistent with the schema is difficult

<table>
<thead>
<tr>
<th>Schema Definition</th>
<th>Schema Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Implicit</strong></td>
<td><strong>Internal</strong></td>
</tr>
<tr>
<td>• High performance</td>
<td>• High performance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Explicit</strong></th>
<th><strong>External</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Highly Extensible</td>
<td>• Easier to manage changes among two or more tools</td>
</tr>
<tr>
<td>• Easily Understood</td>
<td>• Keeping the code consistent with the schema is difficult</td>
</tr>
<tr>
<td>• Well documented</td>
<td>• Difficulties managing changes among two or more tools</td>
</tr>
<tr>
<td>• Check for well formed input is easier to implement</td>
<td>• Tool code is more complicated</td>
</tr>
</tbody>
</table>

**Table 2. Advantages and Disadvantages of Schema Definition and Locality on Exchange**

**Implicit/Internal (✘)** Although an implicit schema definition provides high performance, the fact that the schema definition is embedded in the code means that the capacity of the tool is fixed. Making variations to the code to accommodate different magnitudes of information is difficult.

**Explicit/Internal (✔)** Although the explicit schema definition reduces the performance of the tool, it provides flexibility that makes it easier to adjust the representation to address scalability issues. For instance, one strategy for managing large bodies of information is to exchange only specific pieces of it rather than the whole thing. When the schema definition is explicit, adjusting the amount of information exchanged is much easier than when the definition is implicit.

**Implicit/External (✘)** The implicit schema definition provides high performance but once again the tool is tailored to handle information in a particular way only. Although the schema might be easy to change because it is external, the tool may not be able to handle large volumes of information without significant code changes.

**Explicit/External (✔)** The advantages of this exchange pattern are identical to those for the explicit/internal exchange pattern, although the performance degradation may be more significant because the locality of the schema is external.

**6.3 Simplicity**

_The format is not complex or intricate. This makes it efficient, easier to describe, comprehend, apply and maintain while statistically reducing the prospects for errors and making it easier to process in an automated fashion._ [32]

**Implicit/Internal (✘)** The schema is indeed complex and intricate, being disseminated throughout the code for the tool. It is difficult to understand and maintain making it prone to erroneous modification.

**Explicit/Internal (✔)** The non-embedded nature of the schema specification simplifies the exchange and makes it much easier to understand and maintain. The close proximity of the schema to the tool code provides greater efficiency over the explicit/external exchange pattern.

**Implicit/External (✔✔)** The schema does not exist which simplifies the exchange process and provides an environment where the throughput of information can be maximized.

**Explicit/External (✔)** The schema specification simplifies the exchange process, but its separation from the tool makes it less efficient than the explicit/internal exchange pattern.

**6.4 Neutrality**

_The representation is independent so that as many tools as possible can integrate with it._ [32]

**Implicit/Internal (✘)** There is no neutrality of the representation as it is embedded into the code for the tool.

**Explicit/Internal (✘)** The explicit nature of the schema definition provides a degree of neutrality. Nevertheless, the schema locality is internal to the tool, which impedes the integration of other tools to a standard representation.
Implicit/External (✘) The schema is independent from the tool, which provides it with some degree of neutrality. Nevertheless, the schema is non-existent, so it is difficult to define a standard for other tools to integrate with it.

Explicit/External (✔✔) Neutrality is maximized. The schema definition is completely separate from all tools and is explicitly defined. This makes it easier to integrate other tools to a standard representation.

6.5 Formality

A formal definition reduces the chances for misinterpretation and ensures that it is well understood by all parties. [32]

Implicit/Internal (✘) There is no formal definition of the representation so it is very difficult to transfer knowledge of it to others. This is especially problematic because of the embedded nature of the representation.

Explicit/Internal (✔) By default, the explicit schema definition is formal. Because it is internal to the tool, all concerns relating to the tool implementation are together in the same place.

Implicit/External (✘) The implicit nature of the representation means there is no formal definition. Because the schema locality is external, documentation must be relied upon for information on the representation.

Explicit/External (✔✔) The explicit schema definition is itself a formal means for expressing the structure of the data instances. The external schema locality makes it easier for all tool integrators to understand the representation.

6.6 Flexibility

The format accommodates different tools, languages and syntax for data and schemas. It also accommodates the exchange of incomplete information. [32]

Implicit/Internal (✘) The exchange pattern offers no flexibility at all. The tool itself must handle any accommodation for different tools, languages or data/schema syntax.

Explicit/Internal (✔) The explicit schema provides flexibility for changing the representation. This is partially negated by the fact that the schema is defined internally, which ties it very closely to the tool it is contained in.

Implicit/External (✘) A certain degree of flexibility is offered by the implicit schema definition because it is external from all tools that participate in the exchange. Nevertheless, because the schema definition is implicit, it is difficult to offer representational flexibility. Each tool must conform to the same non-existent schema definition. This tends to force developers to keep to a rigid representational standard.

Explicit/External (✔✔) Flexibility is maximized. First the schema definition is external, so it is not tied to any one tool. Second, the schema is explicitly defined so the representation is clear and easily modified.

6.7 Evolvability

The format can be changed easily to accommodate future needs. [32]

Implicit/Internal (✘) Change is difficult to manage because the representation is embedded in the code for each tool.

Explicit/Internal (✘) Although the explicit schema definition supports evolutionary changes, the internal locality of the schema ties the representation too closely with the tool. Changes to the representation must be implemented on a tool-by-tool basis.

Implicit/External (✘) Although the schema definition is located externally, change is difficult to accommodate because the schema definition is implicit. Evolutionary changes are difficult to implement when all parties involved must approve it.

Explicit/External (✔✔) Evolvability is maximized. The external schema definition does not tie the representation to any one tool. The explicit definition encourages evolutionary change in a collaborative manner.

6.8 Popularity

Adoption of the format is widespread so that as many tools as possible can take advantage of it. [32]

All Exchange Patterns (─ )

Exchange patterns that have external schemas may become more popular because they facilitate the use of well-accepted document exchange methods such as XML. Nevertheless, the success of a particular integration technology ultimately rests with those who use it within the reverse engineering community.
6.9 Completeness

Everything needed to exchange information successfully is included. The user does not have to look after details relating to the exchange. [32]

All Exchange Patterns (—)

We have differentiated between schema and instance data in the exchange process. Although these two components are required to carry out exchange (and in this way they typify how the exchange is managed), they do not represent a complete exchange format.

6.10 Schema Identity

Transformation of instance data while preserving its identity is supported. The integration technology is capable of converting instance data from one schema into instance data of another schema. The instance data remains the same; it is just represented differently from one schema to the next. [32]

Implicit/Internal (✘) The implicit nature of the schema definition makes it very difficult to support transformation of instance data. The use of the representation is embedded into the code for the tool. Identifying instance data and transforming it into an equivalent alternate representation is challenging.

Explicit/Internal (✔✔) The schema definition is explicit which greatly assists in identifying the structure and semantics of instance data. At the same time, the schema is internal so it reflects the tool’s view of instance data. Transformation of this schema to an external schema for exchange is all that is necessary.

Implicit/External (✘) Once again, the implicit nature of the schema definition makes it very difficult to support transformation of instance data. The external schema definition is non-existent, which makes it difficult to identify a transformation to another schema that will preserve the identity of the instance data.

Explicit/External (✔✔) The explicit schema definition lays out the representation in a single location external to the tool. Schema transformation can be carried out from each of the tools participating in the exchange.

6.11 Solution reuse

Wherever possible, use existing techniques and methods with the goal of reducing the amount of time and effort spent in testing and deploying the format. [32]

Implicit/Internal (✘) The representation is embedded into the tool code, which makes it very difficult to reuse.

Explicit/Internal (✘) Although explicitly defined, the representation remains closely tied to the tool. This tool centricity makes it difficult to reuse the representation outside the tool environment.

Implicit/External (✘) The non-existent schema definition is not easily described which makes it difficult to reuse.

Explicit/External (✔) The representation is defined explicitly and is not tied to any one tool. This tends to make it easier to reuse and makes it easier to implement and test.

6.12 Legibility

A human reader can easily understand the format. [32]

Implicit/Internal (✘) The embedded nature of the representation makes it difficult to understand, especially for non-programmers. Well-documented code may partially offset this problem.

Explicit/Internal (✔) Understanding of the representation is much easier when it is explicitly specified in a single location within the tool.

Implicit/External (✘) The non-existent nature of the schema definition impedes understanding of the representation. This combined with the fact that the schema locality is external means that independent documentation must be relied upon to get information about the representation.

Explicit/External (✔) An explicit schema definition eases the legibility of the representation. The external locality of the schema ensures that the representation is tool independent.

6.13 Integrity

Special mechanisms ensure information is exchanged without errors. [32]

All Exchange Patterns (—)

The integrity of the exchange ultimately rests on the underlying technology used to communicate information.

6.14 Comparative Summary

Table 4 summarizes our evaluation of how each exchange pattern satisfies the SEF requirements listed in [32]. It is clear that the use of an explicit schema definition with
Table 4. Exchange Pattern Satisfaction of SEF Requirements

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Implicit/Internal</th>
<th>Explicit/Internal</th>
<th>Implicit/External</th>
<th>Explicit/External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transparency</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Scalability</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Simplicity</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Neutrality</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Formality</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Flexibility</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Evolvability</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Popularity</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Completeness</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Schema Identity</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Solution Reuse</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Legibility</td>
<td>✘</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Integrity</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

external schema locality satisfies all the requirements that relate to exchange. In fact, five of the nine exchange-related requirements are strongly satisfied by the explicit/external exchange pattern. Following a distant second is the explicit/internal exchange pattern. Both of the exchange patterns with an implicit schema definition are the least satisfactory. Between these two, the implicit/external pattern is strongly beneficial solely because of its simplicity.

To summarize our evaluation, the use of schemas with an explicit definition and external locality are the preferred choice for an SEF. The schema definition appears to be the most important factor in the evaluation. Schema locality matters as well, but it is a more minor option to consider.

7 Conclusion

Before tools can exchange information, they must share, at some level, the organization for the data exchanged. That is, they must share a schema. In this paper we examined the various ways in which schemas are represented and used in tools. Schema use was classified according to how and where a schema is defined, leading us to identify four patterns of exchange. We examined these exchange patterns and discussed how each has been used in existing reverse engineering tool integration technologies. An evaluation of each exchange pattern against the requirements for a standard exchange format revealed that explicit/external schemas are the most suitable for integrating reverse engineering tools. Explicit schema definitions enforce schema identity and formality, while external schemas promote neutrality, flexibility and evolvability.

These observations strengthen the case for GXL [15, 16] as a standard exchange format. GXL’s use of explicit/external schemas in combination with a metaschema for E-R graphs provides a common base from which any schema for representing software can be derived. Efforts are currently underway to standardize schemas for C++ [12] and the Dagstuhl Middle Model [24]. As these and other schemas are standardized, GXL will serve as an effective technology for integrating reverse engineering tools.

References


[34] Introduction to TkSee 2.0. http://www.site.uottawa.ca/~tcl/kbre/options/intro.html.


