Integrating Relational Data with Actor-Based Models

Rudolf Schlatte

Institute for Informatics, University of Oslo

Abstract

The Actor model can be a good basis for modeling real-world concurrent and distributed systems. Behavioral modeling languages like ABS are used to create representations of real-world behaviors ("digital twins"). One challenge of creating digital twins is to integrate real-world facts and data that steers or influences the model's behavior. This paper presents a way of integrating relational and other tabular data sources into models written in ABS. The approach has been implemented using the SQLite database engine, but can be adapted for any data source that delivers data in a tabular format.

1 Introduction

Modeling is the process of writing down an abstraction that nevertheless captures relevant aspects of reality. Executable models have semantics similar to programming languages, and can be used to simulate (execute, "run") the model. ABS [2] is an actor-based, executable modeling language. ABS features algebraic datatypes and a functional sub-language, sequential imperative object-oriented programming, and actors communicating via asynchronous messages and cooperative scheduling.

Many models, be it in ABS or other formalisms, are data-driven in the sense that modeling the real world needs some real-world data to guide the model. This data often takes the form of sequences of data values, e.g., a series of numbers representing sensor readings, and is stored in tables such as databases or CSV files [4].

In ABS, the data needed for data-driven models was traditionally expressed as long literal lists in the ABS source, but large data sets exposed limitations in the compiler toolchain. Additionally, we feel that such data is a first-class part of the modeling activity, and should stay in its "natural" form to be accessible for analysis by different tools and not only ABS. Therefore, we implemented a way to query existing SQLite [1] databases from ABS. The query results are expressed as ABS-native datastructures in a running model.

1.1 Related Work

Scientific software packages like Pandas [3] provide facilities to import structured data as *Data Frames*, rich data structures that offer APIs to filter, aggregate, group and otherwise process tabular structures in-memory. Our solution leaves such processing tasks to the SQL query, and only loads necessary data into memory. Another embedded querying solution, working both in-memory and over various external data structures, is LINQ [5]. LINQ queries can be written in programming-language native syntax which is transparently translated to backend-specific queries; we decided to expose the backend query language instead, both for ease of implementation and to make sure all features of a given query backend are directly expressible.

2 Implementation

A design goal was to integrate queries to external data sources as closely into the existing language as possible, i.e., to avoid changes to syntax or type system. Since queries return

constant data (barring concurrent modification of the data source outside of the model), queries are integrated into the functional layer of ABS. The modeler implements each query as a named function returning a list of ABS data values (strings, numbers, or user-defined datatypes). Listing 1 shows such a query.

```
data Person = Person(String name, Int age);

def List<Person> find_persons_with_age(Int min_age) =
  builtin(sqlite3, "test.db",
          "SELECT␣name,␣age␣FROM␣persons␣WHERE␣age␣>␣?",
          min_age);

{
  foreach (p, i in find_persons_with_age(18)) {
    println(‘Person number $i$ is $p$‘);
  }
}
```

Listing 1: Example embedded query—each returned row is converted into a `Person` data value. The main block prints all persons aged 18 or above.

To implement query embedding, the special function body `builtin` was augmented with parameters that are interpreted not as normal ABS code but specially handled by the code generation backend. The first parameter (`sqlite3`) causes a query to be generated against the database named in the second parameter (`"test.db"`). The third parameter is a string literal containing the query text, while all subsequent parameters are ABS expressions whose value will be passed as query parameters. (Query parameters in the SQLite dialect of SQL are written as question marks.)

The data in the rows resulting from executing the query must be type-compatible with the constructor of the datatype returned by the query function. In Example 1, the first column of the query must be convertible into a String and the second one into an Integer, since these are the types accepted by the constructor of the `Person` type.

When calling the function `find_persons_with_age`, the return value can be processed just like any other ABS list, for example, iterated over with `foreach`. Note that queries are only allowed as function bodies and are not standard ABS expressions. This encapsulation makes the approach more amenable to static analysis tools, which is an important design principle of ABS.

2.1 Type Safety

Whenever systems with different data models are connected, mismatch between type systems is a challenge. The approach presented guarantees that whenever an embedded query returns a list of values, these values will be type-correct with respect to the return type of the ABS function implementing the query.[1] The following type errors are detected at run time:

- A mismatch in query parameter count or type(s)
- A mismatch between query result (arity, types elements) and ABS result type

[1] This is the same guarantee as presented by the Model API of ABS (https://abs-models.org/manual/#-the-model-api), which accepts values via HTTP requests.

- A mismatch between stated and actual type of data in the database.

The first and second item can be statically checked at compile-time: some of them (parameter count, result arity) by parsing the SQL query, the others by checking against the database schema. The third cause of errors (type mismatch between schema and value in the database) can only be statically detected by analyzing the database at runtime, and relying on the source data staying unchanged. Note that type errors in the data can occur in some data sources only, but among those are CSV files and the SQLite database engine, two very common data formats.

Since the database schema is unlikely to change, we plan to implement type checks that rely on schema information only. Type-checking the source data against its schema might be implemented together with reading of CSV files, which are often hand-edited and hence prone to contain data errors.

3 Conclusion and Future Work

Behavioral systems of real-world systems often need to integrate with real-world data that specifies aspects of the behavior that should be modeled. By integrating structured data produced by SQL queries into ABS models, we achieved the desired integration of domain-specific data into behavioral models.

The presented solution avoids the scaling and agility problems we encountered when transforming large static tables of data (10,000 and more elements) into ABS code. Such data forms the basis of certain forms of models of real-world occurrences (e.g., machine load measurements over time), and is already available in machine-readable formats. The conversion into ABS added complexity and exposed scalability problems in underlying software components that are not under the author's control; the approach presented in this short paper avoided those problems.

In the future, we plan to integrate more data sources such as CSV files and SPARQL queries. Note that care must be taken when reproducibility of simulations is desired, since the external data source becomes, in effect, part of the model. Therefore, we will focus on data sources that are file-based and can be maintained together with the ABS models and other analysis tools.

References

[1] Richard D Hipp. SQLite 3.36.0, 2021. https://www.sqlite.org/index.html.

[2] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.

[3] The pandas development team. pandas-dev/pandas: Pandas 1.0.3, March 2020. https://doi.org/10.5281/zenodo.3715232.

[4] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, RFC Editor, 2005.

[5] Mads Torgersen. Querying in C#: how language integrated query (LINQ) works. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 852–853. ACM, 2007.