Towards Automated Checking of Component-Oriented Enterprise Applications

Jukka Järvenpää and Marko Mäkelä

Laboratory for Theoretical Computer Science

Helsinki University of Technology

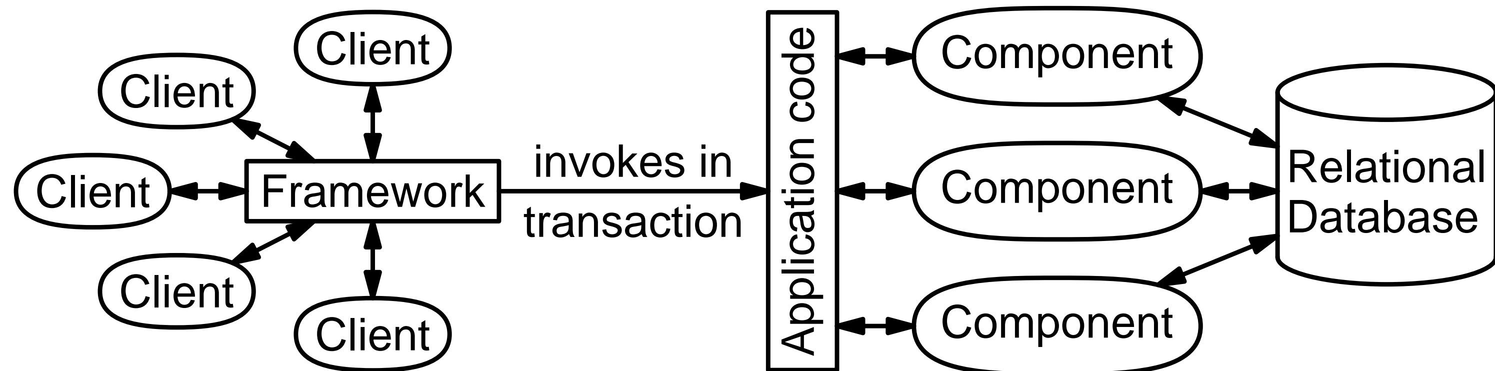
P.O.Box 9205

02015 HUT

Finland

August 26, 2002

The Target Architecture (Java 2 Enterprise Edition)



Component-oriented enterprise applications are:

- built around a *central data store* encapsulated by
- *reusable components* (e.g., Enterprise Java Beans);
- customised via a small amount of *application code* that is
- invoked by an application server *framework* (e.g., a J2EE-enabled HTTP server)

Requirements of Successful Component Deployment

- The components and the framework must adapt to the customer environment,
- the building blocks should have a suitable granularity both from the customer's and the vendor's view (not too complex to manage), and
- the system must work even when third party components are integrated.

Formal description techniques and model checking could be helpful in meeting these requirements.

Applying Formal Methods in Software Engineering

A number of preconditions have to be met in order for model checking to be successful:

- precisely defined software architecture
- high-quality repeatable processes (uniform methods of design and development)
- integrated tool support in the development environment
  - allow developers and programmers to use familiar language
  - be as close to “push-button verification” as possible

Questions to the Verification Engineer

1. Does the interplay of application code and software components conform to the high-level protocols (business processes)?

2. Does the application code use all component interfaces properly? Are all design contracts met?

3. Is the application behaviour in agreement with the data model?

Examples of Design Contract Breaches (Coding Mistakes)

- Do any assertions fail, indicating misuse of an interface?
- Will each transaction eventually be either committed or rolled back?
- Are any run-time errors possible?

Examples of Data Model Violations

- Can the referential integrity rules or row constraints of the database be violated?
- When a new data column is added to a table, existing code dealing with the table must be modified. Does this introduce any errors? Is some modification forgotten?

Two Extremes of Modelling Software Systems

- Model everything by hand
  - weeks or months of work per model, error prone
  - must redo everything whenever the code or the level of abstraction changes
- Write a fully automated model checker front-end
  - either the translator will be unflexible and recognise a subset of the language, or
  - months or years must be invested in writing the translator

The middle course is to model some code automatically and other parts by hand. That requires a modular structure where the hand-modelled parts should remain stable. It is a simple but flexible approach, as the translator can be tailored for the application.

Modelling the Target Architecture: General Observations

- all persistent data is kept in a transactional relational database management system
- the object–relational mapping takes place in the software components
- the application code and the components are strictly single-threaded; there may be multiple database and framework threads

In essence, the systems make use of shared memory multiprocessing.

Modelling the Target Architecture: Main Elements

environment (clients via the framework) initiate requests

transactions encapsulate the processing of requests

application code defines the high-level processing logic; it is invoked by the framework, and it glues components together

components provide an interface between the application code and the database

database provides a persistent data store for the application

Specifying the Desired Properties

- safety properties: “nothing bad happens;” derived automatically
  - data integrity rules
  - assertions in program code
  - safety guards against modelling artefacts (e.g., a table becomes full)
- liveness properties: “something good eventually happens;” provided by the user
  - use *specification patterns* (predefined templates of logic formulae) and
  - provide semi-automated derivation of *fairness assumptions*

Modelling the Environment

- Varying the domains of transaction parameters can greatly affect state space sizes.
- Input validation code can be integrated with the environment: only select such parameters that will not be rejected.
- By default, any transaction can be invoked at any time. Sometimes it is useful to restrict this (e.g., execute initialisation transactions first).

Abstracting the Application Code and the Database

- symmetry reduction of transaction parameters, primary keys and object identifiers
- simple garbage collection: on commit, discard all objects (transient data)
- rolled-back transactions are modelled as deadlocks (no “reset” transitions or events)

- interleaved executions of transactions are eliminated by applying a *resource token* that makes all transactions mutually exclusive:

Modelling the Object–Relation Mapping

The persistent data store uses the relational model, while the run-time data structures are in objects. A mapping between these two worlds is provided by the components.

simple components map one table to a class (generated from a high-level description)

complex components provide an interface to a group of tables (manually written code)

composite components wrap other components together with small amount of code

Simple components and composite components can be automatically translated to formal models, but complex components must be modelled manually (by the component vendor), because they tend to use constructs that a simple translator cannot support.

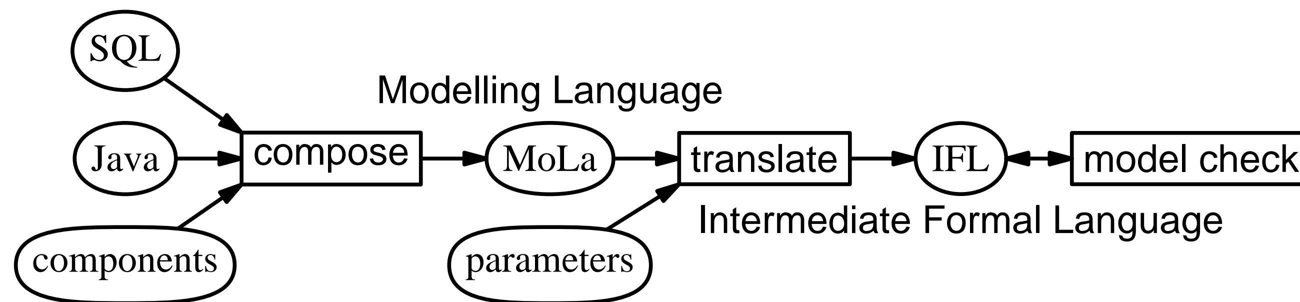
Mapping Objects and Relations in a High-Level Petri Net

object identifiers: A unique identifier or reference is assigned to each created object. These are purged together with all objects when a transaction completes.

existence tests: Normally, a Petri net transition is disabled when something it tries to read does not exist. Implementing an `SQL where` clause requires a modelling trick, such as associating a counter of matching rows for each search criterion.

aggregate operations: For example, when an invoice header is deleted, the invoice lines listing the billed items are deleted as well. In MARIA, such operations can be modelled with a single high-level transition, thanks to quantification.

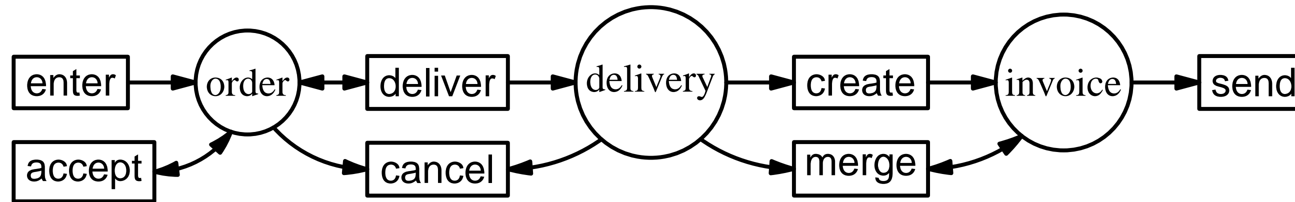
The Proposed Tool



We propose an automated tool that is integrated in the development environment. The tool would be used by software developers who want to see whether the application code they write fulfills the specified high-level requirements of the system.

The prototype tool uses MARIA as a model-checking back-end. It parses database definitions (SQL), application code (Java) and a home-brew language for modelling complex components. The parameters fed to the translator include data domain sizes and some constraints for the behaviour of the environment.

An Example Application: Processing Orders



Above is a Petri-net like view of our sample application. Each rectangle represents an entry point in the application code, something that can be executed in a transaction. There are four important tables (“customer” and “item” can be abstracted away):

order of an item by a customer

delivery of an order

invoice issued to customer, consisting of multiple rows

invoice row belonging to an invoice

Model Checking the Example Application

As a feasibility study, we modelled the system by hand. In the context of our proposed tool, almost all functionality would be captured by simple components.

In our model, there are 12 high-level transitions and 10 places. Six of the places are redundant; the four essential places correspond to the database tables.

A number of data integrity constraints were verified. In the model, each transaction was modelled as a single transition, which would not likely be true for an automated translator.

The model has two kinds of parameters: the sizes of the data domains, and flags for controlling what kind of deliveries can be cancelled.

As the table in our paper shows, the state space size depends more on certain domain sizes than others. Symmetry reduction would clearly be beneficial.

Related Work

- Modelling database applications with Petri nets: NetCASE (1995)
  - pure code generation; difficult for extending an existing system
- Adapting model checking to software environments: VeriSoft (1997)
  - mainly for C and C++; difficult to combine with Java or SQL
- Automated modelling of program code: PathStar, Bandera, SLAM
  - reverse engineering based on abstraction rules; translator must process all code
- Combining automatically derived and hand-made models: Lie et al. (2001)

Conclusions

- High-level software interfaces create an opportunity for applying formal methods.
- Well integrated and automated tool support is needed for adopting model checking techniques in the industry.
- Our proposed tool transforms application code, database schema and a repository of component models into a verifiable model.
- Safety properties can be automatically derived from existing engineering documents.
- Stating and verifying liveness properties is likely to require a verification engineer.
- State space explosion can be tackled by varying the parameters of the models and by implementing reductions (symmetries, partial orders, modular analysis).