Towards Automated Checking of Component-Oriented Enterprise Applications

Jukka Järvenpää Marko Mäkelä[*]
Laboratory for Theoretical Computer Science,
Helsinki University of Technology,
P.O.Box 9205, 02015 HUT, Finland

July 29, 2002

Abstract

Building enterprise applications using component-based frameworks has been suggested as a way to help companies manage their software assets. We propose tool support for managing these high-level data-centric applications with formal methods. Our method is based on extracting a system model from the models of components and from the application code which glues the components together. This model is used for generating state spaces that can be checked for desired or undesired properties. In order to manage the state space explosion problem we propose that the application developer controls some parameters of the model. Even though the insight of the application developer is still needed, we believe that creating tool support for the proposed method could contribute to the success of the component-based approach.

Keywords. software components, transactions, abstractions, verification, Java

1 Introduction

Enterprise application systems have traditionally been used to integrate internal business processes within companies.

The current trend is to expand integration across organisations. The objective is to create more dynamic trading partner relationships, to reduce costs and to increase the productivity of companies participating in a networked economy. This trend sets high demands for companies to maintain and modify their core systems.

Enterprise application systems have often evolved from in-house development projects. The alternative is to buy a packaged solution from an outside software vendor. Compared to a packaged product, an internally developed system could better match the needs of the company. On the other hand, in-house development costs must be carried solely by the company, while software vendors can distribute their costs to a larger number of clients. Also, the package vendor gains experience

[*]This research was financed by the Helsinki Graduate School on Computer Science and Engineering, by Jenny and Antti Wihuri Fund, and by Academy of Finland (Project 47754).

when delivering solutions to different companies, which allows it to incorporate best practice into the package. With internal development this is harder to achieve.

Buying a packaged solution does not come without difficulties either. When the package is installed and configured, the final result can be more determined by the abilities and options of the package rather than the needs of the organisation [12]. Choosing a monolithic package is a commitment that locks the customer into a business relationship with the vendor for a long time. Sometimes this is mutually beneficial, but it could turn out to become harmful if the vendor is not capable of offering the support needed, or goes out of business.

1.1 The Component Approach

A middle course between the "make" and "buy" approaches is to build the system from reusable components. In this approach, the core system contains only minimal functionality, and the necessary tailoring is done by composing distinct encapsulated entities within the system framework. Component-based frameworks are partial implementations that provide fundamental elements, structural integrity and extension points.

Components are packaged software artifacts that provide functionality through a set of well defined interfaces. Component-based systems are expected to become a key business productivity solution for suppliers and consumers in the application market [22]. The anticipated benefit is a flexible and economical infrastructure, where organisations have a considerable choice of procurement to create customised solutions [22]. System acquisition and modifications should also become more manageable, because the modular architecture allows components to be deployed and updated individually [12]. Well-defined interfaces isolate component development from the rest of the system.

Figure 1 gives a simplified picture of a component-based framework application. The picture demonstrates how the framework invokes application code, which extends and refines the framework. The application code acts as glue between the framework and the components. Some of the business rules are contained within the application code, but most program code, such as database access, is hidden behind the component interfaces.

We believe that there is a great demand for tool support for managing applications built using component-based frameworks. This article presents a proposal for extracting models from application code and the components it accesses. These formal models can be explored to check whether the application behaves as required. Many problems must be solved to make such an approach possible. Among other things, the process of extracting a model should be highly automated, and the state spaces generated by the resulting model should at the same time be both manageable and correspond to the implemented behaviour. Last but not least, application coders must be able to specify the system requirements and to see the error traces in terms familiar to them.

1.2 Outline

The rest of this article is organised as follows. Section 2 discusses the economic and environmental preconditions that must be satisfied before software verification can be used for managing component-based framework applications. Section 3 describes an application environment in terms of architecture, software processes and tools that make it possible to extract verifiable models from applications developed in the environment. It also contains a code excerpt from a sample

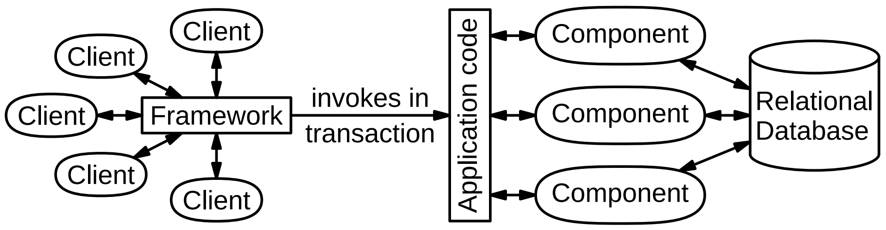


Figure 1: Simplified view of an application in a component-based framework.

application we use to clarify our method. Section 4 defines our modelling framework of enterprise applications at a conceptual level. It describes what kind of questions the model should be able to answer, and it discusses some modelling considerations, which must be taken into account. Section 5 revisits our example and shows how a developer might use the proposed tool. Finally, we discuss some related work and conclude our presentation.

2 Preconditions for Component Software Verification

2.1 Economic and Environmental Preconditions

The component approach, as such, does not guarantee to solve all the problems in enterprise application software management. The components and the framework must be designed to meet industry requirements—not a trivial task at all. Everything must adapt to the customer environment and be manageable by both the customer and the vendor. The integrity and the functionality of the system must be guaranteed even when third party components are integrated. Conventional software engineering practices, such as requirements analysis, system modelling, version control, testing and documentation, retain their importance in component-based system development.

More advanced software engineering techniques, such as automated software verification, could contribute to the success of component-based systems. Applying formal methods to component systems gives a profoundly different starting point for third-party component markets. A formal model—an abstract description of a system—can be thoroughly analysed by computer tools to increase confidence in the system working according to the specification. System models can be derived by composing the high-level application logic with models of the system framework and the components. Verification techniques have the potential to decrease maintenance costs, too. Costs could be saved by simulating or verifying the impact of application changes on a formal model. Automated verification runs could replace some of the otherwise required testing.

2.2 Architecture, Process and Tool Preconditions

To successfully apply verification techniques in industrial-scale application development, the environment has to fulfil a number of requirements:

Precisely defined architecture. As verification is based on a model, the results are meaningful only if the model corresponds to the executable application on an abstract level. This requires that the application structure is precisely defined and implemented.

High quality repeatable processes. In the same way, the correspondence between the model and the application necessitates that the design processes used to create the executable code and the verifiable model are repeatable and of such quality that small deviations in the design process do not lead to substantial differences.

Integrated tool support in the development environment. Constructing verifiable models manually would consume too much time and require highly specialised skill. Automated tool support eliminates these problems as well as errors in translation. Verification tools should accept input directly from the elements created by the developer in the design domain and map the output back to the design domain.

3 The Environment of the Component Framework

3.1 Enterprise Application Architecture

Enterprise applications are data-centric systems where persistent data is stored in databases and processed by application programs. Typically, enterprise applications build upon a client–server architecture where business rules are implemented on the server side, and clients take care of the user interface.

In industrial software packages, databases usually follow the relational model [23, Chapter 2.3]. The conflicts that may arise when several processes access the database simultaneously are resolved using *transactions* [23, Chapter 9]—atomic sequences of operations. Either the effect of all operations are committed to the database, or the whole transaction is rejected.

In a database management system, operations belonging to different transactions are interleaved with each other for performance reasons. In a formal model, the operations of database management system can be abstracted by serialising the transactions, allowing the model to process only one transaction at a time.

3.2 An Example Application: Processing Orders

To gain more insight into component-based enterprise application frameworks, we show an extract from an example application in Figure 2(a). The application code[1] is invoked by the framework when an order is entered. The involved components are shown in the UML diagram of Figure 2(b). The code retrieves customer and item information from the database, updates the order with this information and stores the order into the database.

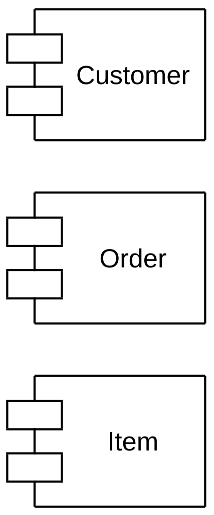
The semantics of the example deserves some additional remarks:

- If the method raises an exception or returns the error code of false, the framework will roll back the transaction, so that no changes are committed to the database.
- The method does not store any internal state between successive calls. The persistent state is kept in the database.
- Most of the implementation is hidden behind component interfaces.

[1]This method could be implemented in the J2EE architecture [21] in a session bean.

```
private boolean process_OrderEntry(Order o)
  throws Exception {
  Customer c = dbCustomer.load(o.getCustomer_id());
  if (c == null) {
    error_notfound(o.getOrder_id(), o.getCustomer_id());
    return false;
  }
  Item i = dbItem.load(o.getItem_id());
  if (i == null) {
    error_notfound(o.getOrder_id(), o.getItem_id());
    return false;
  }
  o.updateOrder(c, i);
  dbOrder.store(o);
  report_ok(o.getOrder_id(), "order entry");
  return true;
}
```

(a) a method implemented in Java



(b) the components

Figure 2: An application for processing orders.

   – The objects dbCustomer and dbItem are simple components, whose load methods simply retrieve objects from the database.
   – The composite component dbOrder hides a more complicated implementation. In this example, we assume that the method store tries to combine the new order with an open order the customer might have. If no such order exists, a new order is stored into the database.

- Some code, such as calls to the logging facilities error_notfound and report_ok, does not affect the state of the application and should be omitted from the model.

3.3 Software Processes

Maintaining a component-based software system requires that repeatable processes be followed to manage the framework, the components, and the application code. The majority of the application lifetime costs are incurred by the maintenance period [20, Chapter 30]. From the customers' point of view, most maintenance tasks are likely to concern application code modifications and occasional deployment of new components.

In order to make application modifications more effective, we propose that automated verification takes place before system level testing. The objective is to gain more insight into the application than could be achieved by pure static analysis techniques. In this step, a system model—derived from the application code and the components—is explored with a verification tool that presents any errors as executions of the application code.

The proposed automated verification step requires that for each deployed component, there is a model of its implementation. In order to guarantee this, both the models and the implementations should be the results of the component design process.

The tools that assist in these processes are described in the following section.

3.4 Tool support

To automate the verification step, we need a tool which parses the application code, accesses a library of component models, composes the parsed application code with the component models and feeds the result to a model checker. This tool should also map any error traces from the model checker back to execution traces of the application code.

To ensure that a component implementation conforms to its model, we propose the following procedures to be aided by tools:

Automated derivation of simple models. Models for simple components could be produced automatically from the same repository information from which the implementations are generated. Examples of such components are object/relational mapping routines, which allow the data in relational databases to be stored and retrieved as objects.

Automated derivation of composite models. When a component is implemented by wrapping other components together with application code, its model can be derived automatically by composing the parsed application code with the models of the wrapped components. This can be accomplished with the same tool that creates system models.

Manually maintained models. Models for the most complex components must be maintained manually. This is tedious, but the involved cost is justified if the component can be sold to several installation sites.

Manual work easily leads to differences between the model and the implementation. Conformance testing [8] could help to locate the errors. The manually constructed component model acts as the specification that the implementation can be formally tested against. Again, conformance testing should be supported by tools.

4 Formalising Component-Based Applications

In order to analyse a system, an automated tool needs a description of both the implemented and the desired behaviour. The system implementation is transformed into a formal model that generates a state space, such as a high-level Petri net. The desired properties are formulated in logic or as automata. Some properties can be derived automatically, others are retrieved from a library or specified by the application developer.

A model of an enterprise application is bound to have a huge number of reachable states. Therefore, the model must be structured and designed carefully. This section describes the main elements of the model and how they relate to the application. It also discusses the properties we would like to extract from the state space graph, and how the model should be built to limit the effects of the state space explosion as much as possible.

4.1 Modelling Elements

The core model can be mapped to shared memory multiprocessing. The shared memory is the database, and the competing processes are the transactions initiated by the environment.

These elements relate to the architecture in Figure 1 in the following way:

Environment. The environment models the application framework and the inputs from the clients. When state space exploration techniques are applied to a model, the model must represent a closed system, which means that the behaviour of the environment must be specified. The environment invokes methods of the application code, initiating a transaction for each request.

Transactions. Transactions model service execution within the application framework. If all operations succeed within the application code and within the components invoked to serve a request, the changes made to the persistent objects are committed to the database. Otherwise, the persistent state remains unchanged as the transaction is rolled back.

Application code. Application code may implement business logic or components, or extend or connect existing components.

To ease the extraction of models, application code is written in a subset of the Java programming language, comprising assignments, conditions, loops and virtual method calls. Some constructs, such as threads, have been deliberately excluded.

Simple components. These are the basic building blocks made available to application developers. Each operation in the component interface is defined with one or more transitions. The model can behave nondeterministically.

Database. The database is the persistent data-store of the application. Operations are grouped in transactions, which can be either committed or rolled back.

4.2 The Properties

Verification or model checking refers to the process of checking whether a model of a system behaves according to its specification. Automating this step requires that both the model and the behaviour requirements are in machine readable format.

Model checking is a useful tool in situations where new functionality is added to the system. Implementing the functionality might require changes to be made in several locations in the application code, and the application coder would like to gain assurance that he has correctly identified these locations.

When a property is violated, the verification tool should report an error trace, an execution sequence leading from the initial state of the system to the error. At the coarsest level, the error trace should display the names and parameters of the components that are executed. Sometimes the user would like to view parts of the trace in more detail, showing individual statements and variables in the application code.

In enterprise applications, many properties can be derived automatically from database definitions and program code. Only high-level requirements need to be formulated interactively.

4.2.1 Safety Properties

Safety properties are requirements on finite executions. Intuitively, they are statements of the form “nothing bad happens”. For example, if a business function requires that a new database field is always initialised in certain business situations, the application coder can phrase rules or assertions such as “Field *x* is set whenever *y* holds.”

Enterprise application databases are most likely designed to contain fields recording status information, such as whether an order has been accepted, or whether it has resulted in a delivery or a sent invoice. Safety properties can express requirements which may refer not only to several such status fields at once, but also to a history of states. This allows us, for example, to verify that the status fields fulfil a requirement such as “if an invoice is sent, a delivery must have occurred and the order must have been accepted.”

Data integrity rules. Many relational database management systems have built-in mechanisms for ensuring the integrity of stored data. It is possible to restrict the set of allowed tuples by defining row constraints (e.g., “the delivery date of an order must be either null or later than the registration date”) or foreign keys (e.g., “each order item row must refer to an existing order”).

Whenever a tuple is inserted, modified or removed, the database management system checks all relevant rules and rolls back the transaction if any rule is violated. The rules form a safety net against errors that may occur in exceptional situations. These rules might never be violated in basic tests, but exhaustive verification will find all violations by testing all possible cases.

Assertions in program code. Many programming frameworks include an assertion facility. The program code may be instrumented with Boolean conditions that reflect the programmer’s assumptions. Rules can be specified for the data passed to or returned by methods, or as arguments to a special “assert” macro that aborts program execution if the specified condition does not hold.

Such assertions can be automatically transformed to safety properties of the model. Similarly to database integrity rules, the assertions are most likely to fail in exceptional situations that can be best found in exhaustive testing.

Identifier pool alert. Section 4.3.2 explains why abstract identifiers are needed in the model and describes our solution for managing the state space explosion problem by using small enough data domains. Deadlocks may occur if these identifiers run out. This is not necessarily an error in the application, but it may be caused by the model where the number of available identifiers is limited. A safety guard can assist the user in managing the identifier domain sizes. When the last identifier is taken, the safety guard is triggered to indicate a potential problem. The occurrence of this event would suggest that the domain should be enlarged. Such checks should be optional.

4.2.2 Liveness Properties

Verifying that a system never reaches an erroneous state is a very powerful way to increase confidence in the correctness of the system. However, sometimes this is not enough, and we want to claim that “something good eventually happens,” such as “an order entered into the system will eventually also be processed.”

A liveness property is violated if there is an infinite execution where progress is not guaranteed. Usually this means that some actions can be repeated infinitely in the system, and the same states are visited again and again.

When expressing liveness properties we need also to assume that certain actions receive fair treatment. When strong fairness is assumed for a transition, it must be executed infinitely often if it becomes enabled infinitely often.

In our example, where orders are entered and processed separately, we must assume that neither transaction is neglected in order to verify that each order eventually results in a delivery.

Some of the more complicated application behaviour requirements can only be specified by the designer, who expects the application to behave in a certain way. This task can be eased by providing the designer with *specification patterns* [6], templates of formulae or property automata.

4.3 Modelling Considerations

We shall now consider the modelling elements from Section 4.1 in more detail.

4.3.1 The Environment

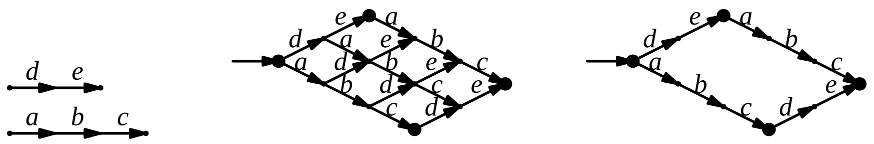
Domains of transaction parameters. The domains of transaction parameters greatly affect the number of reachable model states. Validated input is stored into a database, which can become quite large. This behaviour is reflected in the model so that enlarging the input domains result in even larger state spaces. In order to manage the state explosion problem, we have to limit the domains. When the application developer is allowed to select the input domain sizes individually, he can check different aspects of the system. Obviously this approach relies on the intelligence of the user and does not prove the absence of errors. However, checking a restricted model might reveal errors more easily than testing or simulating a more complete model.

Automatic unification of transaction parameter domains. The application code is statically analysed to identify the relations between database fields and transaction parameters. Each group of related fields and parameters is assigned an own domain. Developers cannot be assumed to keep such mappings up to date, as the system is maintained over a long period of time by different persons. Unifying the domains is essential for models with scalable domain sizes.

Controlling transaction invocations. One way to attack the state space explosion is to guide the search by restricting the behaviour of the environment. For instance, transactions for filling in basic information could have priority over the actual processing transactions. One way to arrange this is to divide the behaviour of the environment into phases where only certain transactions will be invoked. Formally, the environment can be defined as a finite automaton whose actions are labelled with transactions.

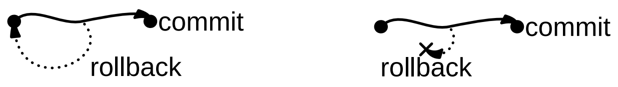
4.3.2 The Database and the Transactions

Initialising the database. In the initial state of the model, the database is empty. The model generates all the possible database states allowed by the application logic, as the environment nondeterministically initiates transactions.



(a) the transactions (b) interleaved executions (c) resource token abstraction

Figure 3: The effect of a resource token on scheduling two transactions.



(a) a true model (b) utilising deadlocks

Figure 4: Modelling a rolled back transaction. Solid arrows denote committed transactions that lead from one persistent state to another. A rollback (dotted arrow) leads back to the originating persistent state, or to an artificial deadlock.

Symmetry reduction of transaction parameter domains and object identifiers. Identifier values model objects references in the application code and surrogate keys in the database, such as item numbers. Symmetry reduction [14] can lead to exponential savings by exploiting the fact that the actual values of these identifiers are irrelevant.

Static analysis can determine the set of operations performed on each domain. Symmetry reduction is only compatible with assignment and equality test. For instance, integer arithmetics requires a (limited) domain of integers or equivalence classes.

For each identifier domain, the model contains a pool of available values.

Transactions and resource tokens. Since the database management system isolates transactions from each other, the transactions can be modelled to be mutually exclusive. This can be arranged by introducing a *resource token* [11] that must be "possessed" by the active transaction.

Figure 3 illustrates the effect of a resource token. There are two enabled transactions, consisting of 2 and 3 operations. Depending on the order in which the operations in the transactions are performed, the system will follow different paths to the final state, shown rightmost in Figures 3(b) and 3(c). Only the corner states of the depicted lattices are *persistent*, meaning that the database is in a committed state. Some of the *transient* states have been eliminated in Figure 3(c).

The resource token abstraction may interfere with partial order reductions [9]. Those techniques work best when the processes in the system are as independent as possible. The resource token makes all transactions depend on each other. Also, verifying liveness properties requires a strong fairness assumption for the first transition of each transaction and a weak fairness assumption for the transitions that return the resource token. The model checker algorithm in MARIA [15] manages these assumptions in an efficient way.

Rolling back transactions. When a transaction is rolled back, the requested changes to the persistent data must be ignored. This can be accomplished in two ways (Figure 4):

(a) by restoring the persistent data from a back-up copy, or

(b) by setting a "rollback" flag that disables all transitions in the model—an artificial deadlock.

Translating rolled back transitions to deadlocks simplifies both the model and its state space. In a real system, rolling back a transaction should restore the database to its original state, as depicted in Figure 4(a). In exhaustive state space enumeration, all reachable states of the system are considered, and deadlock states pose no problem. The search algorithm can still distinguish genuine deadlock states of the system from these artificial deadlocks by examining the “rollback” flag.

4.3.3 Components and Application Code

Mapping objects to relations. There are two types of data in enterprise applications. The transient data that is being processed is managed in objects, while the persistent data in the database is stored as tuples from relational calculus. The models of the components that provide mappings between tuples and objects must address the following issues:

object identifiers: Compared to the relational data model, the object model adds a level of indirection in the form of object identifiers. A unique identifier or reference is assigned to each created object. When an object is no longer needed, the identifier can be freed. The dynamic allocation of identifiers can lead to a combinatorial explosion unless some reduction techniques are applied. Our model limits the explosion by purging all objects and identifiers upon entering a persistent states.

existence tests: Databases are often tested for the existence of records. For instance, the component dbOrder introduced in Section 3.2 must determine whether the customer has an open order, and place a new order if necessary. In Petri nets, transitions are enabled if enough items exist in their input places. Defining an action for the case when something is absent requires a modelling trick, such as using a complement place or a counter, or reserving a special value for denoting absent items.

aggregate operations: Sometimes it is necessary to perform an operation on a group of data, such as all items that belong to an invoice. The total invoiced amount is the sum of the prices of the ordered items multiplied by the ordered quantities. When an invoice header is deleted, the invoice lines listing the billed quantities and identifying the items are deleted as well. This kind of operations can be modelled in high-level nets by making use of inhibitor arcs, as Billington demonstrates [3, Chapter 8], or by introducing auxiliary attributes that can be used to limit dynamic quantifications in the MARIA net class [16]. For instance, there could be a derived place that maps invoice identifiers to invoice line counts.

Components and their composition. Component services can be modelled as transitions that define the effect of invoking the service interface. Nondeterminism can be modelled by defining conflicting transitions for a service. We call this kind of model elements *simple components*.

Transitions can be difficult to derive automatically, if the logic of the program code is complicated. This limits the use of simple components. More complicated cases can be maintained manually as discussed in Section 3.4. Another possibility is to create composite component models. They are derived automatically from the application code. Each statement in the application code is assigned a program counter value within the composite component. A statement corresponds to a transition that performs a computation step and updates the program counter.

Composite components allow program logic to be extracted automatically from the application code. The program counter values increase the state space, even though the counter is reset when the transaction is completed. However, this information is relevant when mapping an error trace to application code statements. The source code file names and line numbers can be encoded either in enumerated program counter values or in transition names.

Simple components do not need program counters. Thus, they can be composed with the rest of the application model by transition substitution. Modelling component execution with a single transition does not introduce intermediate states in the same way as using a program counter does.

Path compression and nondeterministic choices. Eliminating interleavings with the resource token, as illustrated in Figure 3, can result in some non-branching state sequences in the state graph. Such sequences can be collapsed by applying path compression [17].

Nondeterministic components and conditions within application code introduce branches in the state space. The branch target states cannot be eliminated by path compression. However, MARIA is able to distinguish "visible" and "hidden" states. Only the visible states, corresponding to the persistent states of the model, need to be permanently stored.

Eliminating input validation code with static analysis. Typically, application code validates its input. Nearly half the code in Figure 2(a) deals with erroneous input. This code can be omitted from the formal model if the environment is constrained in such a way that it sends only such parameter combinations to the method that would pass the validation. This may lead to significant reductions at the cost of additional static analysis.

Method calls. Object-oriented programs typically contain a large number of method calls. When a *virtual method* is called, the run-time system must determine the type of the object and *dispatch* the call to the applicable method. Sometimes the call target can be determined at compilation time.

The translation of virtual method calls can be simplified by generating a dispatcher method for each virtual method. The dispatcher contains a switch block that branches according to the type of the object. In each branch, the dispatcher jumps to a method of a derived class. In this way, each virtual method invocation can be implemented as a non-virtual call to a dispatcher procedure.

Method calls involve some overhead of storing return addresses and copying parameters. For short methods, it is more efficient to substitute calls to the method with the program code in the method body. This technique is referred to as *inlining*. It can eliminate trivial intermediate states, but it may also produce significantly bigger models. In essence, it is a tradeoff between the model size and the number of reachable states.

Folding. Some entities can be modelled as a single high-level Petri net place or as a collection of simpler places. The choice whether to fold may affect the space and time requirements of state space enumeration. Folding places adds flexibility to transitions.

For instance, when the control flow of a program is modelled with a single high-level "program counter" place, a switch statement can be translated into a single transition that jumps to one of the case labels. If there was a separate program counter place for each statement in the program, the program flow might be more clearly visible from a graphical presentation of the net, but translating the switch statement would require more transitions, in fact one for each case label.

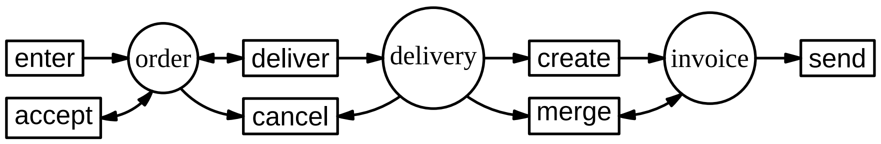


Figure 5: An abstract view of an order processing application.

Similar choices can be made in data type definitions. When a class hierarchy is translated to a single data type definition, objects of a base class can be stored in the same place, no matter which derived class it belongs to. Defining separate data types for derived classes requires a set of places (and transitions) for each derived class.

5 Analysing the Example Application

To evaluate the feasibility of the presented approach, we manually constructed a high-level Petri net model for our example application that was introduced in Section 3.2.

Figure 5 presents a simplified view of the main information flows of the application as a Petri net like graph. The processing starts when an order is entered into the system. Deliveries are controlled by a separate system, to which the order processing system sends a delivery request message, once the order has been accepted.

The delivery system informs the order processing system of completed deliveries. Either system may also initiate a procedure to discard the order and the delivery request.

A delivery confirmation message is transformed into an invoice that will be sent later. If there is an unsent invoice for the customer who made the order, the delivery is merged with this invoice. The last step in the processing chain is to send the invoice to the customer.

5.1 The Model of the Demo Application

In the generated model of the application, each transaction comprises a simple component. Since there are no program counters, all reachable states of this model are persistent database states.

This model was hand crafted, and some abstractions were made. Most notably, the database tables “customer” and “item” were eliminated, because they do not control the behaviour of the transactions we are interested in.

The implementation of the application contains functions for entering and updating information that does not control the application logic, such as names, addresses and prices. Without loss of generality, the domains of these data fields were restricted to one value, which essentially removes the fields from the formal model.

The “order” table contains, among others, three columns for quantities: the quantity of ordered items, the quantity of delivered items, and the quantity of items that have been invoiced. The last column is redundant, as its data can be derived from deliveries and invoices. Databases sometimes contain redundant information, either because deriving the information is computationally too expensive or because the data used for deriving the information might be cleaned up later from the live database to a data warehouse system. Such redundancy could be detected in static analysis, which may be expensive. On the other hand, eliminating redundant fields does not reduce the number of reachable states, but the space needed for representing a state.

Invoices are stored in two tables. The “invoice row” table links deliveries to the header table “invoice.” In the implementation, the invoice rows are numbered, so that invoices can be retrieved in a consistent order. While the order of invoiced items may be relevant in printed documents, it does not matter in our formal analysis. Therefore, the row number column was abstracted away.

The resulting model in MARIA format [18] has 12 transitions and 10 places. Four places correspond to the modelled database tables. The markings of the remaining six places are functions of the database contents. Three places are identifier pools of unassigned order, delivery and invoice numbers and one place counts the lines belonging to each invoice. Two places—which would be connected to the transitions create and merge depicted in Figure 5—indicate which customers have unsent invoices and which do not.

5.2 A Usage Scenario

In this example scenario, an application coder wants to verify that all the referential integrity rules are respected, and that an order entered will eventually be processed. Processing an order means that the order is delivered and invoiced, or it is cancelled.

The referential integrity rules are translated into safety properties, and the liveness requirements are specified in LTL. Both are checked on the fly by the MARIA tool.

The application designer is likely to begin the analysis of the model by assigning all data domains the cardinality 1. In this configuration, some transactions are permanently disabled. For instance, the transition merge of Figure 5 cannot be enabled unless there may be multiple orders and deliveries. MARIA can detect and report dead transactions.

Next, the user might want to enlarge some domains in order to enable more behaviour in the model. Increasing the cardinalities may reveal spurious errors. For instance, when the database accepts multiple orders but only has room for one invoice, it will be impossible to invoice all deliveries unless they can be combined to the single invoice.

Verifying high-level liveness properties is an interactive procedure where the domain sizes, fairness assumptions and the environment need to be adjusted if an unjustified error is reported.

5.3 Some Results

As Table 1 shows, the state space of the model grows significantly when any of the domains is enlarged. Some of the growth is inherent in the application, as discussed in Section 4.3.1, but much of it is due to the lack of symmetry reduction in the tool we used. Because the system behaviour does not depend on actual data values, exploiting symmetries could lead to exponential savings.

Some domains have a greater impact on the state space size than others. If the system accepts at most one order, it does not matter much how many customers there are who can place the order or how many items are available to be ordered. But as soon as there can be multiple orders and deliveries, the state space explosion breaks loose.

In Table 1, not all parameters of the system are varied. Orders are never cancelled, and the database has room for only one invoice. The system has a large state space, and only parts of it can be viewed at a time. When one parameter is incremented, other parameters must be limited and some transactions may need to be disabled. Obviously, not all errors can be guaranteed to be

Table 1: Sizes of reachability graphs generated by the model without and with path compression reduction when at most one invoice can be generated and orders cannot be cancelled. Increasing the cardinalities of orders and deliveries (O), customers (C) or items (I) affects the numbers of reachable states $|V|$ and transition occurrences $|E|$.

O=1		Original		Reduced	
C	I	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
1	1	16	19	7	11
1	2	31	46	13	29
1	3	46	81	19	55
1	4	61	124	25	89
2	1	43	58	25	41
2	2	85	148	49	113
2	3	127	270	73	217
2	4	169	424	97	353
3	1	82	117	46	82
3	2	163	306	91	235
3	3	244	567	136	460
3	4	325	900	181	757
4	1	133	196	73	137
4	2	265	520	145	401
4	3	397	972	217	793
4	4	529	1,552	289	1,313

O=2		Original		Reduced	
C	I	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
1	1	427	986	409	1,003
1	2	1,609	4,616	1,537	4,591
1	3	3,547	12,042	3,385	11,915
2	1	2,665	7,376	2,521	7,279
2	2	10,369	38,432	9,793	37,759
2	3	23,113	106,992	21,817	105,263
3	1	8,227	26,118	7,741	25,595
3	2	32,329	145,368	30,385	142,703
3	3	72,307	419,958	67,933	413,531

O=3		Original		Reduced	
C	I	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
1	1	14,680	49,341	14,518	50,809
1	2	107,983	447,870	106,687	451,345
2	1	194,923	794,226	192,331	798,889
2	2	1,496,197	8,197,284	1,475,461	8,175,073

found in this kind of analysis, but even partial verification has better coverage than testing. None of the data integrity rules built in the model are violated in the combinations we checked.

6 Related Work

Modelling database systems with Petri nets is nothing new. One earlier method is NetCASE [19], a Petri net based computer aided software engineering (CASE) technique that covers everything from requirements analysis to code generation. It may be hard to apply this kind of methods in practice, where things tend to be built on top of existing systems. We believe in automated reverse engineering, the opposite of code generation.

The PathStar project at Bell Labs [10] showed that a programming language can be treated as a formal model, provided that the source code is annotated appropriately for an automated translator that makes suitable abstractions. In that project, verification experts translated requirement specifications from English prose to LTL and maintained the abstraction rules of the translator, so that it was possible to model check the software under development on a daily basis.

The Bandera [5] and SLAM [2] toolkits create abstract verification models from source code. Bandera inputs the abstractions from the user, while SLAM iteratively refines them by itself. Neither tool seems to support the composition of derived models with hand-crafted fragments.

Lie et al. [13] present a method for automatically extracting models from low level software implementations. The extracted model is combined with a model of the hardware. Their approach

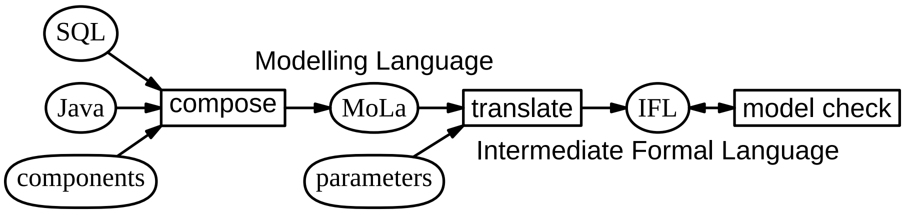


Figure 6: A block diagram of the proposed tool. The prototype will be based on MARIA, but our Intermediate Formal Language can be easily interfaced with other model checkers.

is similar to ours, except that we combine models extracted from high level program code with abstract models of software components.

7 Conclusion and Future Work

Component based software systems are expected to create a flexible and economical infrastructure where companies have a considerable choice of procurement to create customised solutions. When components can be deployed and updated individually, system acquisition and modifications should become more manageable than before. With a simple example, we demonstrated how these data-centric applications are constructed and what their environment looks like.

The architectural style of component-oriented applications, where functionality is hidden behind high-level interfaces, creates an opportunity for applying formal methods, such as state space analysis. Our approach is based on extracting a formal system model from the models of software components and from the application code which glues the components together. This model is formally checked for desired or undesired properties.

Adopting advanced software engineering techniques, such as model checking, in an industrial setting requires well integrated and automated tool support. We propose a tool that allows software maintainers to verify the correctness of systems before system level testing. The objective of this verification step is to gain more insight than could be achieved by pure static analysis techniques.

This tool, depicted in Figure 6, transforms application code, database schema and a repository of component models into a verifiable model of the system. Many desired properties of the system are derived automatically from database definitions and assertions in the application code. Some safety guards, such as the identifier pool alert, are optional. Verifying high-level liveness properties is likely to be an interactive procedure, where the user is required to control the fairness assumptions and the model parameters, such as input domain sizes, if an unjustified error trace is reported. If errors are found, they are presented in terms of the application code.

The application behaviour is mapped to a formal model based on shared memory multiprocessing. In the model, the shared memory is the database and the competing processes are the transactions initiated by the environment. The structure of the application is restricted in such a way that the relations between transaction parameters and database contents can be derived automatically. Each group of related fields and parameters is assigned an own domain.

The state explosion problem is tackled from two directions. Primarily, we rely on the user managing the parameters of the model. Secondly, we build the model in such a way that state space reduction can be accomplished in verification tools.

The state explosion problem can be alleviated by keeping the data domains small. Minimising the data domains could result in some of the application behaviour missing from the model. Here we rely on the user insight and allow him to individually select the sizes of various data domains. The developer may also specify how the environment should behave: which transactions should be invoked and in which order. In this way, users can generate state spaces revealing different aspects of the application behaviour. This partial verification resembles testing, but it can have better coverage.

Our modelling framework abstracts from the inner workings of database management systems. Only one database transaction is processed at a time. Ideally, we would like to store only the persistent database states and the transitions between these states. The state explosion can also be attacked with symmetry reduction [14]. It relies on the fact that the actual values of identifiers are irrelevant, as long as only assignments and equality tests are applied to them. These conditions can be checked by the tool that constructs the verifiable model.

We believe that the proposed tool could help in reducing application maintenance costs. Savings are possible if some of the otherwise required testing can be substituted with verification runs. Applying formal methods to component systems gives a profoundly different starting point for third-party component markets. A formal model—an abstract description of a system—can be thoroughly analysed by computer tools to increase confidence in the system working according to the specification. Without such confidence, customers are easily locked in ordering all further development from the original system vendors.

This article describes "work in progress." Sections 4 and 5 were mainly written by the second author, while the idea of applying state space analysis to component-based software originated from the first author who is preparing his licentiate's thesis on the subject. His plans include writing a front-end for the MARIA tool [16] and using it in simulated application maintenance work. If the results are positive, it will be most interesting to find industrial applications and to see how our approach could be augmented by modelling the business processes [1] and database performance [4]. Also, conformance testing of components [8] could be implemented in the framework.

Acknowledgements

The authors would like to thank Charles Lakos—especially regarding the identifier pool alert—and Nisse Husberg and the anonymous referees for their comments and suggestions.

References

[1] Wil M. P. van der Aalst. Making work flow: On the application of Petri nets to business process management. In Esparza and Lakos [7], pages 1–22.

[2] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, Toronto, Canada, May 2001. Springer-Verlag.

[3] Jonathan Billington. Extensions to coloured Petri nets and their application to protocols. Technical Report 222, University of Cambridge, Computer Laboratory, Cambridge, England, May 1991.

[4] Ing-Ray Chen and Rajakumar Betapudi. A Petri net model for the performance analysis of transaction database systems with continuous deadlock detection. In Hal Berghel, Terry Hlengl and Joseph Urban, editors, *Proceedings of the 1994 ACM symposium on Applied computing*, pages 539–544, Phoenix, AZ, USA, March 1994. ACM Press, New York, NY, USA.

[5] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM Press, New York, NY, USA.

[6] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, Los Angeles, CA, USA, May 1999. IEEE Computer Society Press, Los Alamitos, CA, USA.

[7] Javier Esparza and Charles Lakos, editors, *Application and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002*, volume 2360 of *Lecture Notes in Computer Science*, Adelaide, Australia, June 2002. Springer-Verlag.

[8] Leonard Gallagher. Conformance testing of object-oriented components specified by state/transition classes. Draft technical report, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, USA, April 6, 1999.

[9] Patrice Godefroid, Doron Peled and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, July 1996.

[10] Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11:65–79, 2001.

[11] Nisse Husberg and Tapio Manner. Emma: developing an industrial reachability analyser for SDL. In Jeannette M. Wing, Jim Woodcock and Jim Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 642–661, Toulouse, France, September 1999. Springer-Verlag.

[12] Kuldeep Kumar and Jos van Hillegersberg. ERP experiences and evolution. *Communications of the ACM*, 43(4):23–26, 2000.

[13] David Lie, Andy Chou, Dawson Engler and David L. Dill. A simple method for extracting models from protocol code. *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001*, pages 192–203, Göteborg, Sweden, July 2001. IEEE Computer Society.

[14] Tommi Junttila. Finding symmetries of algebraic system nets. *Fundamenta Informaticae*, 37(3):269–289, February 1999.

[15] Timo Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In José-Manuel Colom and Maciej Koutny, editors, *Application and Theory of Petri Nets 2001, 22nd International Conference*, volume 2075 of *Lecture Notes in Computer Science*, pages 242–262, Newcastle upon Tyne, England, June 2001. Springer-Verlag.

[16] Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In Esparza and Lakos [7], pages 427–436.

[17] Marko Mäkelä. Efficiently verifying safety properties with idle office computers. In Charles Lakos, Robert Esser, Lars M. Kristensen and Jonathan Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 11–16, Adelaide, Australia, June 2002. Australian Computer Society Inc.

[18] Marko Mäkelä. Maria model of the Jive demo application. `http://www.tcs.hut.fi/maria/samples/jive/`.

[19] Thomas Marx. NetCASE—a Petri net based method for database application design and generation. Research report 11-95, University of Koblenz, Germany, September 1995.

[20] Roger S. Pressman. *Software Engineering: A Practitioner's Approach, European Adaptation*. McGraw-Hill International Editions, 2000.

[21] Bill Shannon. *Java 2 Platform, Enterprise Edition: Platform and Component Specifications*. Addison-Wesley, 2000.

[22] David Sprott. Componentizing the enterprise application packages. *Communications of the ACM*, 43(4):63–69, 2000.

[23] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems. Volume I: Classical Database Systems*. Computer Science Press, 1988.