Automated Extraction of Abstract Behavioural Models from JMS Applications

Elvira Albert[1], Bjarte M. Østvold[2], José Miguel Rojas[3]

[1] Complutense University of Madrid, Spain
[2] Norwegian Computing Center, Norway
[3] Technical University of Madrid, Spain

Abstract. Distributed systems are hard to program, understand and analyze. Two key sources of complexity are the many possible behaviors of a system, arising from the parallel execution of its distributed nodes, and the handling of asynchronous messages exchanged between nodes. We show how to systematically construct *executable models* of publish/subscribe systems based on the Java Messaging Service (JMS). These models, written in the executable Abstract Behavioural Specification (ABS) language, capture the essential parts of the messaging behavior of the original Java systems, and eliminate details not related to distribution and messages. We report on JMS2ABS, a tool that automatically extracts ABS models from the *bytecode* of JMS systems. Since the extracted models are formal and executable, they allow us to reason about the modeled JMS systems by means of tools built specifically for the modeling language. For example, we have succeeded to apply simulation, termination and resource analysis tools developed for ABS to, respectively, execute, prove termination and infer the resource consumption of the original JMS applications.

1 Introduction

Reverse engineering is a key technique to understand and improve software that is available only in executable form. In this paper we focus on reverse engineering, or decompilation, of distributed Java applications given in *bytecode* form, with the purpose of increasing the understanding of such applications though analysis of reverse-engineered executable specifications. In the context of mobile code, programming languages which are compiled to bytecode and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET. The execution model based on virtual machines has two important benefits when compared to classical machine code. First, bytecode is *platform-independent*, i.e., the same compiled code can be run on multiple platforms. Second, since the virtual machine is not directly executed on the hardware, it is possible to apply a *sandbox* model which guarantees that the bytecode does not have access to certain assets of the platform unless the code is explicitly granted access to them. In languages such as Java and C#, handling bytecode has a much wider application area than handling source code since the latter is often not available.

We study a specific class of distributed systems called *publish/subscribe* systems [9]. For this class of systems resilience, scalability and performance are common desirable properties, and a key part of such systems is a special *middleware* for message communication, which ensures such properties [14,21]. Furthermore we focus on applications built using the Java Messaging Service (JMS) [13], an industry-standard technology for realizing publish/subscribe enterprise systems in Java. Our goal is to extract *abstract behavioral specifications* that capture the essentials of the messaging behavior, eliding implementation details, but preserving enough behavior that analysis can draw conclusions about distribution and resource consumption of the original systems. The modeling language, called abstract behavioral specification language (ABS) allows to abstract from implementation details: Abstract data types and functions specify internal, sequential computations, while concurrency and distribution are handled using *active objects*. Analysis of ABS models is supported by a set of research tools.[4]

We report on JMS2ABS, a tool which automatically extracts an ABS model from a JMS application in bytecode form. The main phases of the extraction process are: (1) Decompile the bytecode into a higher-level intermediate representation with structured control flow. (2) Based on annotations added by the programmer, generate an ABS model from the intermediate representation. (3) During generation, insert calls to a pre-written ABS library of the JMS middleware, in order to model publish/subscribe middleware behavior.

The main contributions of our work can be summarized as follows:

- Section 4 provides a general and system-independent model of a subset of JMS publish/subscribe systems;
- In Section 5, we define a procedure for translating the code of a JMS publish/subscribe system into an executable model, and realize this as a tool;
- Section 6 applies existing tools developed for the ABS language in order to draw conclusions about the systems;
- Finally, Section 7 reports on a prototype implementation of our approach and evaluates it on two JMS examples.

2 Publish/Subscribe Communication in JMS

JMS is an industry standard for message communication in Java enterprise systems [13]. It offers APIs for configuring message passing services and for performing the message passing (i.e., encode, send, receive, and decode messages). One may realize various kinds of messaging systems using JMS; we focus on publish/subscribe systems.

Fig. 1 provides an overview of the publish/subscribe programming model of JMS. *Subscribers* have the ability to express interest in events or messages in order to be later notified of any message generated by a *publisher* that matches their registered interest. The basic model for publish/subscribe interaction relies

[4] These tools are currently being developed by the ongoing EU project HATS (FP7-231620), http://www.hats-project.eu.

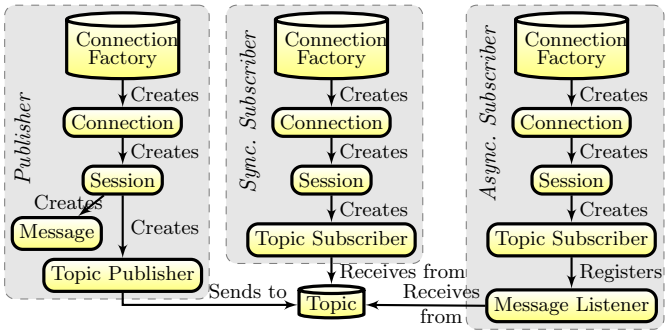

Fig. 1. Overview of the JMS publish/subscribe programming model.

on a message notification service (middleware) to provide storage and management for subscriptions, to mediate between and decouple publishers and subscribers, and to deliver messages efficiently. Such middleware manages addressable message destination objects denoted as *topics*. The steps that *publishers* and *subscribers* perform, as depicted in Fig. 1, are: (1) Discover and join a topic of interest by means of a *connection factory* of topics. (2) Establish a *connection* to the factory and then start a new *session* for a such connection. (3) Create a *topic subscriber* for the session which allows receiving (subscribers) and sending (publishers) messages related to the topic of interest. (4) Create and publish a message (publisher). (5) Receive a message (subscriber).

We consider the subset of JMS components depicted in Fig. 1, capturing the essence of the publish/subscribe communication model. In addition, a model of a JMS system must include the state information and logic that decides how messages are processed and exchanged. Features such as transactions or failure recovery are outside the scope of this paper. Fig. 2 shows an example of a JMS publish/subscribe implementation of a basic fruit supply business model, consisting of a `FruitSupplier` class that acts as a publisher of updates for topic `"PriceLists"`; `SuperMarket` class implements asynchronous updates receipts from the topic, time-decoupled (i.e., non-blocking) from the publisher; and `Example` class provides the `main` method that initializes instances of `FruitSupplier` and `SuperMarket`.

Note that the different components are created and retrieved by invoking API methods. In particular, `ConnectionFactory` and `Topic` objects can be either created dynamically or found using JNDI services[5]. Subscribers can retrieve messages either asynchronously using a `MessageListener` object or synchronously through the (blocking) `receive` method of a `TopicSubscriber` object.

3 ABS: A Distributed Modeling Language

Within the OO paradigm, there are two main approaches to concurrency: (1) thread-based concurrency models (like those of Java and C#) are based on threads which share memory and are scheduled preemptively, i.e., threads can

[5] http://www.oracle.com/technetwork/java/jndi/

```
class FruitSupplier extends Thread {
  void run() {
    fac = new TopicConnectionFactory();
    con = fac.createTopicConnection();
    ses = con.createTopicSession(...);
    topic = ses.createTopic("PriceLists");
    publisher = ses.createPublisher(topic);
    message = ses.createObjectMessage(priceList);
    publisher.publish(message);//execution continues
    con.close(); } }
class SuperMarket extends Thread implements MessageListener {
  PriceList priceList;
  void onMessage(ObjectMessage m) {
      newPriceList = m.getObject();
      updatePrices(newPriceList); }
  void updatePrices(PriceList l) {
      Product p;
      for (int i = 1; i <= l.length(); i++) {
        p = l.get(i);
        if (priceList.contains(p)) priceList.update(p);
        else priceList.insert(p); } }
  void run() {
    fac = new TopicConnectionFactory();
    con = fac.createTopicConnection();
    ses = con.createTopicSession(...);
    topic = ses.createTopic("PriceLists");
    subsc = topicSession.createSubscriber(topic);
    subsc.setMessageListener(this);
    con.start();//execution continues
    con.close(); } }
class Example {
  void main(...) { new SuperMarket().start();

new FruitSupplier().start(); } }
```

Fig. 2. Excerpt of implementation of publish/subscribe in JMS.

be suspended or activated at any time. To prevent threads from undesired interleavings, low-level synchronization mechanisms such as locks have to be used. Experience has shown that software written in the thread-based model is error-prone, difficult to debug, verify and maintain [22]. (2) In order to overcome these problems, the *active objects* model [22,16,8] aims at providing programmers with simple language extensions which allow programming concurrent applications with relatively little effort. The common idea is to take advantage of the inherent concurrency implicit in the notion of object in the following way: a concurrent object, conceptually, has a dedicated processor and it encapsulates a local heap which is not accessible from outside the object. Active (also called concurrent) objects operate similar to actors [12] and Erlang processes [5].

ABS [15] is the *abstract behavioral specification* language for distributed concurrent objects that we use to define the models. ABS has a *functional sub-language* with abstract data types and functions to specify internal, sequential computations. The functional language is a standard strict functional language (the details are elsewhere [15]). As regards the concurrent imperative part, the central concept is the notion of component object group (COG), which generalizes the notion of concurrent or active object [12]. Intuitively, each COG has a dedicated *processor* and the COG is a concurrently running, isolated component. A COG can be considered as a container for objects. Its state is a heap of objects which are owned by the COG for their entire lifetime. The behavior

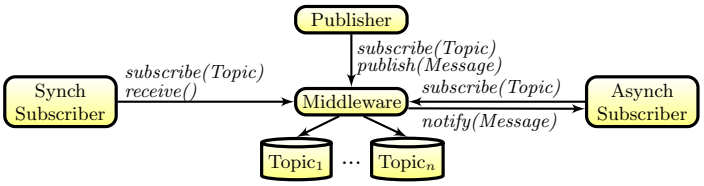

Fig. 3. Concurrent entities in ABS models (counterpart of Figure 1 for JMS).

of a COG consists of a set of cooperative tasks, which, again, are owned by the COG for their entire lifetime.

All communication is via asynchronous method calls between named objects, typed by interfaces. Method calls may be seen as triggers of concurrent activity, spawning new activities (so-called *processes*) in the called object without transferring control from the caller. The method caller may decide at runtime when to synchronize with the reply from a call. In general, an object may have many method activations competing to be executed. Among these, at most one process (or *task*) is active and the other processes are suspended in a process pool. Process scheduling is non-deterministic and occurs only at processor release points. This means that switching between tasks of the same object happens only at specific scheduling points during program execution, which are explicit in the source code and can be syntactically identified. This particular feature of process scheduling makes machine analysis notably simpler (when compared to the thread-based concurrency model).

In ABS syntax, asynchronous method calls are denoted $o!m(\overline{e})$. After asynchronously calling $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object (typed by an interface), and $\overline{e}$ are expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in ABS. First, a return test $x?$ evaluates to false unless the reply to the call can be retrieved. Second, the return value is retrieved by the expression x.`get`, which blocks all execution in the object until the return value is available. The statement sequence $x = o!m(\overline{e})$; $v = x$.`get` encodes a blocking, *synchronous call*, abbreviated $v = o.m(\overline{e})$, whereas the statement sequence $x = o!m(\overline{e})$; `await` $x?$; $v = x$.`get` encodes a non-blocking, *preemptable call*.

4 Modeling Publish/Subscribe Systems in ABS

This section shows how to model the behavior of a publish/subscribe system implemented using JMS by means of the ABS language. The model abstracts away implementation-related details of a distributed Java application while still capturing the essence of cooperation among the components of the system. Our goal is to preserve all essential application properties concerning distribution and performance but improve on clarity and tractability for automatic analysis purposes. Our starting point is the JMS system of Fig. 1, whose model in ABS is shown in Fig. 4 (in the model, *updatePrices* is a function that will be defined later). In particular, we focus on the components that participate in the system (Sec. 4.1) and the operations that can be executed (Sec. 4.2). Sec. 5 will then describe how to automate the model extraction process.

JMS instructions	Equivalent ABS models
Create a distributed object	

`obj = new C();`
`//class C implements Runnable`
`//interface or extends Thread`
	`obj = new cog C();`

Establish new session	

`f = new TopicConnectionFactory();`
`c = f.createTopicConnection();`
`s = c.createTopicSession(...);`
`t = new Topic("TopicName");`
	`s = middleware.createSession();`
	`t = middleware.createTopic("TopicName");`

Send a message	

`pub = s.createPublisher(t);`
`connection.start();`
`m = s.createTextMessage();`
`m.setText("message");`
`pub.publish(m);`
	`pub = s.createPublisher(t);`
	`m = "message";`
	`pub!publish(m);`

Receive a message synchronously	

`sub = s.createSubscriber(t);`
`connection.start();`
`m = topicSubscriber.receive();`
	`sub = s.createSubscriber(t);`
	`Fut<Message> f = s!receive();`
	`message = f.get;`

Receive a message asynchronously	

`sub = s.createSubscriber(t);`
`l = new TextListener();`
`sub.setMessageListener(l);`
`connection.start();`
	`sub = s.createSubscriber(t);`
	`l = new MessageListener();`
	`sub.setMessageListener(l);`

Table 1. Example mapping from Java/JMS to Distributed ABS.

4.1 Distributed Entities

Our ABS model creates only one concurrent object per participant in the distributed communication, namely publishers, subscribers and the middleware; see Fig. 3. Thus, each concurrent entity in ABS encapsulates the behavior of several JMS components that will communicate with the remaining entities by means of asynchronous calls and future variables.

Clients: Publishers and Subscribers. A JMS system relies on a number of objects in order to perform distributed operations. This design makes JMS portable and interoperable across multiple messaging products. However, it often makes the resulting programs harder to understand and thus analyze. In the ABS models we simplify this into a smaller set objects. Namely, a publish/subscribe client in ABS will just need to create a session object and a publisher/subscriber object to interact with the middleware.

Middleware. In a real publish/subscribe system the middleware (the message-oriented middleware, or MOM) is a highly distributed entity. In our model of JMS we simplify the middleware to one central entity. While not a desirable choice in an actual implementation, it still allows to analyse many properties of applications. The middleware entity relies on the concurrency model of ABS to provide publish/subscribe services. The `main` block of the ABS model creates the initial configuration of the publish/subscribe system, see Line 20 of Fig. 4 (the counterpart of Lines 32–33 of Fig. 2). Observe that the model uses COGs to represent each of the distributed/concurrent entities.

```
1  class FruitSupplier(Middleware mw) {
     Unit run() {
3      Topic topic = "PriceLists";
       TopicSession session = mw.createSession();
5      TopicPublisher publisher = session.createPublisher(topic);
       ObjectMessage message = new ObjectMessage(priceList);
7      session!publish(message);
     } }
9  class SuperMarket(Middleware mw) implements MessageListener {
     Unit onMessage(ObjectMessage m) {
11     newPriceList = m.getObject();
       priceList = updatePrices(newPriceList);
13   }
     Unit run() {
15     Topic topic = "PriceLists";
       TopicSession session = mw.createSession();
17     TopicSubscriber subscriber = session.createSubscriber(topic);
       subscriber!setMessageListener(this);
19   }
   { Middleware mw = new cog Middleware();
21   new cog SuperMarket(mw); new cog FruitSupplier(mw);
   } //main block
```

Fig. 4. Extracted ABS model for the running example.

4.2 Operations

Here we consider the operations of a publish/subscribe system.

Message Sending. A publisher sends a message to the topic using a session, see method `run` of class `FruitSupplier` (Lines 3–10 of Fig. 2). The asynchronous semantics of the operation can be simulated by an ABS asynchronous method call, see Lines 3–8 of Fig. 4. In JMS, the sending operation implies some decisions regarding delivery mode, priority and time-to-live for the message. These configuration parameters can be global to a message publisher or specific for each message. For flexibility, we use the latter option and include configuration parameters as properties of messages.

Asynchronous Message Receipt. Method `run` of class `SuperMarket` in Fig. 2 shows that asynchronous message receipt in JMS is achieved by instantiating the `MessageListener` class. The new object is bound to the subscriber object and is able to receive and process incoming messages in its `onMessage` method (named `notify` in the publish/subscribe literature [9]). This method is triggered from the JMS provider upon arrival of a new message to the topic (Lines 23–30 of Fig. 2). In ABS, an equivalent asynchronous message receipt is implemented in Lines 15–19 of Fig. 4. The concurrent behavior, i.e., the interaction with different topics simultaneously, is achieved by sharing the single-threaded session object among clients within the same COG. A serial order of outgoing and incoming messages is implicitly modelled when using a shared session object. Table 1 summarizes the mappings that we have described along this section for our particular example.

5 Automatic Extraction of ABS Models from JMS

Figure 5 provides an overview of the main steps performed by JMS2ABS for automatically extracting ABS models from JMS publish/subscribe systems. The tool receives as input the *bytecode* associated to the JMS publish/subscribe system

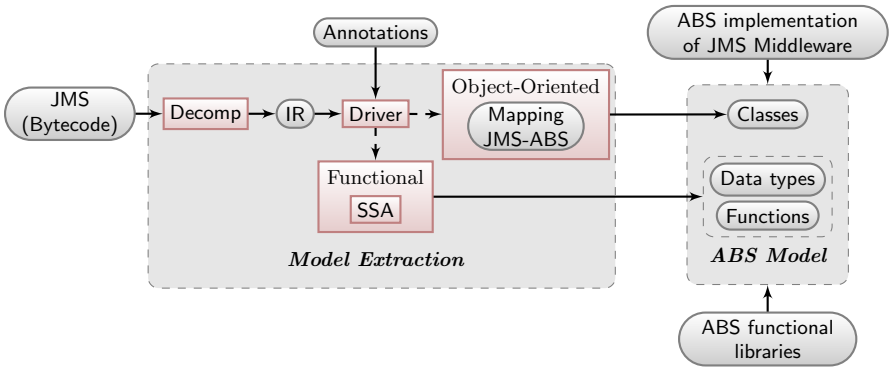

Fig. 5. Overview of main components of JMS2ABS.

and, optionally, a set of *annotations* that indicate which methods of the code should be transformed into functions and which ones into imperative methods. The absence of annotations brings about a purely imperative translation. Intuitively, the following stages are carried out by the extraction process. First, the bytecode is decompiled into a higher-level intermediate representation (IR) which, among other things, features structured control flow. Then, a driver module reads the IR and the set of annotations and directs the model extraction process either towards a functional implementation or towards an imperative one. The extraction of functional code requires a static single assignment (SSA) transformation [4] and automatically generates abstract data types and functions. The ABS library functions include standard data types for lists, trees, etc. and some common functions on these types. They are used by the translation when possible. The imperative object-oriented extraction is based on the modeling of JMS using ABS defined in Table 1. As an external component to this process, we have available the ABS implementation of the specific JMS middleware in use. As a result of the process, an ABS model is obtained which includes abstract data types, functions and classes. The following sections describe in detail the main components of JMS2ABS.

5.1 From Bytecode to Intermediate Representation

A method m in a Java (bytecode) program is represented by a set of *procedures* in the IR such that there is an entry procedure named m and the remaining ones are intermediate procedures invoked only from m. The translation of a program into the IR works by first building the control flow graph (CFG) from the program, and then representing each block of the CFG in the IR as a rule. The process is identical to that of Albert *et al.* [2], hence, we will not go into the details of the transformation but just show the syntax of the transformed program. A *program* in the IR consists of a set of *procedures* which are defined as a set of (recursive) rules. A procedure p is defined by a set of *guarded rules* which adhere to the following grammar:

$$rule ::= p(\bar{x},\bar{y}) \leftarrow g, b_1, \ldots, b_n \qquad g ::= true \mid exp_1\ op\ exp_2 \mid \mathsf{type}(x, C)$$
$$exp ::= x \mid \mathsf{null} \mid n \mid x{-}y \mid x{+}y \mid x{*}y \qquad op ::= > \mid < \mid \leq \mid \geq \mid = \mid \neq$$
$$b ::= x{:=}exp \mid x{:=}\mathsf{new}\ c \mid x{:=}y.f \mid x.f{:=}y \mid q(\bar{x},\bar{y})$$

where $p(\bar{x}, \bar{y})$ is the *head* of the rule; $\bar{x}$ (resp. $\bar{y}$) are the input (resp. output) parameters; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\bar{x}, \bar{y})$ a call-by-value procedure call. The IR supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, C)$, which succeeds if the runtime class of x is exactly C. A class C is a finite set of fields with either numeric (integer) or reference (class name) type.. The key features of this representation, which will simplify the transformation later, are: (1) input and output parameters are explicit variables of rules, (2) *recursion* is the only iteration mechanism, (3) *guards* are the only form of conditional, and (4) objects can be regarded as records, and the behavior induced by dynamic dispatch is compiled into *dispatch* rules guarded by a type check.

```
0: iconst_1                                    updatePrices([this,l],[]) ← i := 1,
1: istore_3                                        rule_2([this,l,i],[]).
2: iload_3                                     rule_2([this,l,i],[]) ←
3: aload_1                                         length([l],[s1]),
4: invokevirtual length:()I                        rule_7([this,l,i,s1],[]).
7: if_icmpgt 43                                rule_7_1([this,l,i,s1],[]) ←
10: aload_1                                        i ≤ s1,
11: iload_3                                        get([l,i],[p]),
12: invokevirtual get:(I)LProduct;                 exists([this,p],[s2]),
15: astore_2                                       rule_21([this,l,p,i,s2],[ip]).
16: aload_0                                    rule_7_2([this,l,i,s1],[]) ←
17: aload_2                                        i > s1.
18: invokevirtual exists:(LProduct;)Z
21: ifeq 32                                    rule_21_1([this,l,p,i,s1],[]) ←
24: aload_0                                        s1 = 0,
25: aload_2                                        add([this,p],[]),
26: invokevirtual updatePrice:(LProduct;)V         rule_37([this,l,i],[]).
29: goto 37                                    rule_21_2([this,l,p,i,s1],[]) ←
32: aload_0                                        s1 ≠ 0,
33: aload_2                                        updatePrice([this,p],[]),
34: invokevirtual add:(LProduct;)V                 rule_37([this,l,i],[]).
37: iinc 3, 1                                  rule_37([this,l,i],[]) ←
40: goto 2                                         ip := i +1,
43: return                                         rule_2([this,l,ip],[]).
```

Fig. 6. Pretty-printed IR for method `updatePrices` of class `PriceList`.

As an example, let us consider method `updatePrices` in Fig. 2. The left-hand column of Fig. 6 shows the bytecode of this method (which is the input to JMS2ABS) and the right-hand column contains the IR that JMS2ABS uses which features the three first points above. We can observe that instructions in the IR have an almost one-to-one correspondence with bytecode instructions (*rule_7* in the IR corresponds to the CFG block starting at bytecode instruction

7, for example), but they contain as explicit parameters the variables on which they operate (the operand stack is represented by means of variables). Another important aspect of the IR is that unstructured control flow of bytecode (i.e., the use of `goto` statements) is transformed into recursion and loop conditions become `guards`, as in rules *rule_2* and *rule_37* for instance.

5.2 From IR to Functional and Distributed ABS

To generate functions from a set of procedures in the IR, JMS2ABS performs three main steps: (1) An SSA transformation on the IR guarantees that variables are only written once [4]. (2) Then, for each recursive rules in the IR, it generates an associated function with the same name, where each instruction is transformed into an equivalent one in the functional sub-language of ABS. The process is similar to decompilation of bytecode to a simply typed functional language [17], to TRS [18] or to CLP programs [11]. Hence, we do not go into the details of the process but rather show an example. (3) Finally, JMS2ABS generates definitions of the data types involved in the functions. This is done by recursively inspecting the types of the class fields until reaching a primitive type, and using data constructors to group the fields that form an object.

The following function corresponds to the bytecode in Fig. 6. It is extracted from the above IR in a fully automatic way. ABS's *let* and *case* expressions, resp., are used to represent variable bindings and conditional statements in the original program. Moreover, observe that several data types declarations have been generated from class `PriceList`. The new algebraic data type `PriceList` has two data constructors: one for the empty list (`EmptyPriceList`) and one for the combination of a product and another list (`ConsPriceList(Product,PriceList)`).

```
  //Data type declarations
2 type ProductID = Int;  type Price = Int;
  data Product = EmptyProduct | ConsProduct(ProductID,Price);
4 data PriceList = EmptyPriceList | ConsPriceList(Product,PriceList);
  //Function definitions
6 PriceList updatePrices(PriceList l) {
    let Int i = 1 in loop(priceList,newPriceList,i);
8 }
  PriceList loop(PriceList l1, PriceList l2, Int i) {
10   let n = length(l2) in
       case i <= n {
12       True  => let p = get(l2,i) in
                  case (contains(l2,p)) {
14                  True  => return loop(update(l1,p),l2,i+1);
                    False => return loop(add(l1,p),l2,i+1);}
16       False => return l1;}
  }
```

All procedures which have not been transformed into functions will become methods of the ABS models. Each ABS class will have as attributes the same ones as in the original Java program. Then, the translation of each method is performed by mapping each instruction in the IR into an equivalent one in ABS. The instructions which involve the distribution aspects of the application are translated by relying on the mapping of Table 1.

Fig. 7 shows the IR for the Java method `SuperMarket.run`. Observe how instructions in lines 2–5 match with the pattern for session establishment shown

in Table 1. Instructions in lines 7–9 correspond to the asynchronous receiving of a message. From this IR it is straightforward to extract the model for method `run` showed in Fig. 4. Because of the correspondence between the involved operations in JMS and ABS, the main properties of the JMS systems (e.g., those regarding *reliability* and *safety* [6]) are preserved.

```
0 run([this],[]) ← tConFac := null, tCon := null, tSes := null, topic := null,
1                  tSubscriber := null, tListener := null,
2                  tConFac := new TopicConnectionFactory,
3                  createTopicConnection([tConFac],[tCon]),
4                  createTopicSession([tCon],[tSes]),
5                  createTopic([tSes,this.topicName],[topic])
6                  createSubscriber([tSes,topic],[tSubscriber]),
7                  tListener := new PriceListener,
8                  setMessageListener([tSubscriber,tListener],[]),
9                  start([tCon],[]), close([tCon],[]).
```

Fig. 7. Pretty-printed IR for method `run` of class `SuperMarket`.

6 Using the ABS Toolset on the Extracted Models

The final goal of the extraction of ABS models from bytecode systems is to be able to perform machine analysis of JMS systems via their equivalent ABS models. This section outlines the application of two ABS tools: the simulator [15] and the COSTABS termination and resource usage analyzer [1].

6.1 Simulation

Once compiled, ABS models can be run in a simulator. The ABS toolset has two main simulators, with corresponding back-ends in the ABS compiler: One simulator is defined using rewriting logic and the Maude system [7], and the other is written in Java. The Maude simulator allows modellers to explore the model's state-space declaratively and model check it. The Java simulator does source-level simulation, meaning that modellers can follow the model's control flow at the statement level and observe object or method state. Both simulators allow modellers to control scheduling of methods, for example, control when a JMS message is sent and when it is received.

6.2 Resource and Termination Analysis

Resource analysis (a.k.a. cost analysis) aims at automatically inferring bounds on the resource consumption of programs statically, i.e., without having to execute the program. The inferred bounds are symbolic expressions given as functions of its *input data sizes*. For instance, given a method `void traverse(List l)`, an upper bound (e.g., on the number of execution steps) can be an expression on the form *l*200+10*, where *l* refers to the size of the list `l`. The analysis guarantees that the number of steps of executing `traverse` will never exceed the amount

inferred by analysis. COSTABS [1], a COSt and Termination analyzer for ABS, is a system able to prove termination and obtain *resource usage bounds* for both the imperative and functional fragments of ABS programs. The resources that COSTABS can infer include termination, number of execution steps, memory consumption, number of asynchronous calls. Knowledge of the number of asynchronous calls is useful to understand and optimize the distributed behavior of the application (e.g., to detect bottlenecks when one object is receiving a large amount of asynchronous calls). COSTABS allows using *asymptotic* (i.e., big O complexity) notation for the results of the analysis and obtain simpler cost expressions.

Method	#Instructions	Memory	#Async Calls
`run`	`max`(allSubscribers)	`max`(allSubscribers)	1
`onMessage`	m*(m+`max`(priceList))+m^2	`max`(priceList)	1

Table 2. Resource analysis results.

Let us analyze the resource consumption of the methods of class `SuperMarket` from the extracted ABS model. Table 2 shows the asymptotic results that COSTABS computes. The upper bound on the number of instructions inferred for method `run` depends on the number of clients that are subscribed to the topic (field `allSubscribers` of class `TopicSession`). `max`(f) denotes the maximum value that field f can take. This is because in our current implementation the size of the list of subscribers is not statically known, as it is updated when a new subscriber arrives (the analysis uses `max`(allSubscribers) to bound its size). As regards the analysis of `onMessage`, it requires analyzing `updatePrices` which traverses the new list of prices `priceList` and, for each of its elements, it checks whether it already exists or must be added to the local list of prices. The latter requires inspecting the object message `m` which is an input parameter of the method. Hence, we obtain a quadratic complexity on the sizes of `m` and `priceList`. The memory allocation accounts for the creation of the functional data structures. Namely, in method `run` (resp. `onMessage`), we create the data structure `allSubscribers` (resp. `PriceList`). Finally, it can be observed that both methods perform a constant number of asynchronous method calls, hence the rightmost column shows a constant complexity (denoted by 1). A main novelty of COSTABS, which is not available in other systems, is the notion of *cost centers*. This is motivated by the fact that distribution does not match well with the traditional monolithic notion of cost which aggregates the cost of all distributed components together. Albert *et al.* [1] propose the use of cost centers to keep the resource consumption of the different distributed components separate.

The cost bounds that are shown in Table 2 are computed as a monolithic expression which accumulates the resources consumed by all objects together. More interestingly, COSTABS can show the results separated by cost centers. In particular, we consider that all objects of the same class belong to the same cost center (i.e., the share the processor). Now, the execution of method `SuperMarket.run` performs steps in three cost centers, namely in `SuperMarket`, `Middleware` and in `TopicSubscriber`. By enabling the cost centers option, COSTABS shows

that `max`(allSubscribers) is the upper bound on both number of instructions and memory in the cost center `Middleware`. In cost center `TopicSubscriber`, the upper bounds on number of instructions and on memory consumption are constant. Also, in cost center `SuperMarket`, the upper bound for both cost models is constant. Method `onMessage` is integrally executed in the `SuperMarket` cost center (hence the same results of Table 2 are obtained). Performing cost analysis of a distributed system, using cost centers, allows detecting bottlenecks if one distributed component (cost center) has a large resource consumption while siblings are idle most of the time.

7 Prototype Implementation

JMS2ABS can be used on 32-bit Linux systems through a command-line interface, is open-source and can be downloaded from `http://tools.hats-project.eu/`. Also, available from the same place, is our ABS model of JMS middleware, examples of how to write publish/subscribe ABS models using the middleware model, and Java/JMS example applications from which models may be extracted. These examples correspond to the running example of this paper, and a Chat example borrowed from Richards *et al.* [20] and slightly simplified. The Java code is accompanied by the necessary Java/JMS libraries and a makefile which may be used to run the tool on the Java examples. Although still a research prototype, JMS2ABS is reasonably efficient. For instance, on an Intel(R) Core(TM) i5 CPU at 1.7GHz with 4GB of RAM running Ubuntu Linux 11.10, the overall time to extract the model for the running example is 910 msec. This time is divided into the time for building the CFG (240 msec.), generating and optimizing the intermediate representation (40 msec.) and building and refining the ABS model (630 msec.). The Chat example is smaller and its overall model extraction time is 790 msec. In this case, the most costly phase is also the model generation and refinements, which takes 490 msec. of the overall time.

8 Related Work

Reverse engineering higher-level specifications from complex third party or legacy code has applications in analyzing, documenting and improving the code. Broadly speaking, we can classify reverse engineering tools into two categories: (1) When the higher-level specification is some sort of software visualization formalism which abstracts away most of the program semantics (e.g., UML class diagrams, control flow graphs or variable data flow graphs), reverse engineering is usually applied in order to understand the structure of the source code faster and more accurately. This in turn can detect problems related to the design of the application, to task interactions, etc. (2) When the higher-level specification provides an abstraction of the program semantics, but still the properties of interest are observable on it, reverse engineering can be used to develop analysis tools that reason about the original code by means of analyzing the reverse engineered specification. This has the advantage that, instead of analyzing the complex

original code, we develop the tools on a simpler formalism which allows inferring the properties of interest more easily.

Our work falls into the second category. The overall motivation behind our work is to be able to analyze (complex) distributed Java JMS applications by means of tools developed for (simpler) ABS models. In particular, we have been able to apply simulation and *cost analysis* techniques developed for ABS programs [3,19] to reason on JMS applications. It is widely recognized that publish/-subscribe systems are difficult to reason about, and there are several previous approaches to modeling their behavior using different formalisms. Baldoni et al. [6] provide one of the first formal computational frameworks for modeling publish/subscribe systems. The focus in this work is different from ours; their main concern is the notion of time in the communication, which allows them to evaluate the overall performance, while we do not consider this aspect. Another formalism for publish/subscribe system is provided by Garlan et al. [10]. Instead of building executable programs as we do, they rely on a finite state machine that can be checked using existing model checking tools.

9 Conclusions and Future Work

Our goal is to show that it is possible to build a tool that automatically extracts useful models for complex distributed systems such as the JMS publish/subscribe using the concurrency and distribution mechanisms provided by the ABS modeling language. These mechanisms are not very different from those used by other distributed object-based modeling languages [12], and so we expect our study to provide useful conclusions beyond the mere case study performed.

Publish/subscribe systems come in a large range of flavors, depending on applications and requirements [9]. The common idea is to asynchronously decouple publishers from subscribers. In a purely centralized model such as the one used in this paper, providing the expected service is not hard, as the server has full knowledge to ensure that messages are sent only to active subscribers, in the same order in which they come in. In general, however, a reusable and general model must allow for decentralized implementations in which full consistency (in the sense that messages are received by only and all subscribers at any given time) and order preservation (same order of messages for all subscribers) cannot be achieved with good performance. We are currently examining ways in which the ABS framework can be extended to allow richer families of implementations.

References

1. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *APLAS 2011*, volume 7078 of *LNCS*. Springer, 2011.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413, January 2012.

3. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In *FM 2011*, volume 6664 of *LNCS*. Springer, 2011.
4. A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33:17–20, April 1998.
5. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.
6. R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito. On the modelling of publish/subscribe communication systems. *Concurr. Comput. : Pract. Exper.*, 17:1471–1495, October 2005.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude—A High-Performance Logical Framework*. Number 4350 in Lecture Notes in Computer Science. Springer, 2007.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *ESOP 2007*, volume 4421 of *LNCS*. Springer, 2007.
9. P. Eugster, P. Felber, R. Guerraoui, and AM. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.
10. D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *SPIN 2003*. Springer-Verlag, 2003.
11. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51:1409–1427, October 2009.
12. P. Haller and F. Sommers. *Actors in Scala: Concurrent programming for the multi-core era*. Artima, PrePrint edition, March 2011.
13. M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. *Java Message Service Specification*. Sun Microsystems, Inc., Version 1.1. April 2002.
14. P. Jiang, J. Bigham, E. L. Bodanese, and E. Claudel. Publish/subscribe delay-tolerant message-oriented middleware for resilient communication. *IEEE Communications Magazine*, 49(9):124–130, 2011.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*. Springer-Verlag, 2012.
16. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.
17. S. Katsumata and A. Ohori. Proof-directed de-compilation of low-level code. In *ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*. Springer, 2001.
18. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *RTA 2010*, volume 6 of *LIPIcs*, 2010.
19. HATS Project. Report on Resource Guarantees, March 2011. Deliv. 4.2 of project FP7-231620 (HATS), available at http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable42.pdf.
20. M. Richards, R. Monson-Haefel, and D. A. Chappell. *Java Message Service - Creating Distributed Enterprise Applications (2. ed.)*. O'Reilly, 2009.
21. K. Sachs, S. Appel, S. Kounev, and A. P. Buchmann. Benchmarking publish/subscribe-based messaging systems. In *DASFAA Workshops 2010*, volume 6193 of *Lecture Notes in Computer Science*, pages 203–214. Springer, 2010.
22. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *ECOOP 2010*, volume 6183 of *LNCS*. Springer, 2010.