Integrating Aspects of Software Deployment in High-Level Executable Models

Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa
Dept. of Informatics, University of Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

Abstract

Software today is often developed for deployment on different architectures. In order to model and analyze the consequences of such deployment choices at an early stage in software development, it seems desirable to capture aspects of low-level deployment concerns in high-level models. In this paper, we propose an integration of a generic cost model for resource consumption with deployment components in timed ABS, an abstract behavioral specification language for executable object-oriented models. Deployment components reflect resource-restricted deployment scenarios, and are parametric in their allocated resources. The cost model may be adapted to specific resources such as concurrent processing capacities or memory. The approach is demonstrated on an example of a web shop with a cost model for concurrent processing resources. We use our simulation tool to analyze system response time for given usage scenarios, depending on the amount of resources allocated to the deployment components.

1 Introduction

Software systems often need to adapt to different deployment scenarios: *operating systems* adapt to different hardware, e.g., the number of available processors; *virtualized applications* are deployed on a varying number of (virtual) servers; and *services on the cloud* need to adapt dynamically to the underlying infrastructure. Such adaptability raises new challenges for the modeling and analysis of component-based applications.

In general abstraction is seen as a means to reduce complexity in a model [19]. In formal methods, the ability to execute abstract models was initially considered counter-productive because the models would become less abstract [11, 13]. Specification languages range from design oriented languages such as UML, which are concerned with structural models of architectural deployment, to programming language close specification languages such as JML [6], which are best suited to express functional properties. For the kind of properties we are interested in here, it is paramount that the models are indeed executable in order to have a reasonable relationship to the way the final code can be expected to work. *Abstract executable modeling languages* are found in between these two abstraction levels, and appear as the appropriate abstraction level to express deployment decisions because they abstract from concrete data structures in terms of abstract data types, yet allow the faithful specification of a system's control flow and data manipulation. Recently abstract executable models have gained substantial attention and also been applied industrially in many different domains [23].

This paper was presented at the NIK-2011 conference; see http://www.nik.no/.

The abstract behavioral specification language ABS [7, 16] is such an executable modeling language with a formally defined semantics and a simulator built on the Maude platform [8]. ABS is an object-oriented language in which concurrent objects communicate by asynchronous method calls and in which different activities in an object are cooperatively scheduled. In order to reflect deployment choices in a model, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language. In recent work, we have extended ABS with time and with a notion of *deployment component* in order to abstractly capture resource restrictions related to deployment decisions at the modeling level. This allows us to observe, by means of simulations, the performance of a system model ranging over the amount of resources allocated to the deployment components, with respect to execution capacity [17, 18] and with respect to memory [2]. This way, the modeler gains insight into the resource requirements of a component, in order to provide a minimum response time for given client behaviors. In the approach of these papers, the resource consumption of the model was fixed by the ABS simulator (or derived by the COSTA tool [1]), so the only parameter which could be controlled by the modeler was the capacity of the deployment component.

In this paper, we refine our previous work by taking a more high-level approach to the modeling of resource usage, and by proposing a way for the modeler to explicitly associate resource costs to different activities in a model. This allows simulations of performance at an earlier stage in the system design, by further abstraction from the control flow and data structures of the system under development. In contrast to previous work, we consider a generic cost model $\mathscr{M}$ and introduce an explicit statement for resource consumption in ABS. The specification of resource usage according to $\mathscr{M}$ may be decomposed from high-level specifications of resource usage to more fine-grained specifications associated with the control flow in the model as the model is refined. Resource costs may be specified by means of user-defined expressions in ABS; e.g., depending on the size of the actual input parameters to a method in the ABS model. Given a model with explicit cost specifications, we show how our simulation tool for ABS may be used for abstract performance analysis of formal object-oriented models, to analyze the performance of the model depending on the resources allocated to the deployment components. The proposed approach and associated tool are illustrated on an example of a web shop with a cost model for concurrent processing resources.

Paper overview. Section 2 gives an introduction to the ABS modeling language, Section 3 presents the modeling of deployment decisions by means of deployment components and cost statements. Section 4 presents a prototype simulation tool for ABS with our proposed extension and and Section 5 presents an application of this tool to an example. Section 6 discusses related work and Section 7 concludes the paper.

2 The ABS Language

ABS is designed to model distributed systems that communicate by exchanging messages. It consists of a functional part to define and modify user-defined datatypes, and an object-oriented, imperative part that models distributed asynchronous communication between active objects. This section gives a brief introduction to the core ABS language, a more in-depth look is contained in the book chapter at [7] and the formal semantics in [16].

Datatypes and Functions. ABS has some basic datatypes: numbers, strings, and Boolean values, with the usual operators, e.g., arithmetic and comparison operators for the `Int` type and logical operators for `Bool`. New datatypes can be defined as follows:

```
data Item = Nothing | Something(Int);
data List<A> = Nil | Cons(A, List<A>);
```

Functions are defined at top level:

```
def Int max(Int a, Int b) = if a > b then a else b;
def Int length<A>(List<A> list) = case list {
  Nil => 0;
  Cons(_, rest) => 1 + length(rest);
};
```

Classes and Interfaces. ABS models are structured into classes. Each class implements zero or more interfaces. Object references are typed by interfaces, not by class.

```
interface Chest {
  Unit putIn(Item i);
  Item retrieve();
}
class LossyChest implements Chest {
  Unit putIn(Item i) { skip; }
  Item retrieve() { return Nothing; }
}
```

Objects fields are always initialized with a value, and parameters are directly accessible as fields, so many traditional uses of constructors are unnecessary in ABS. The optional *init block* can be used for more complicated object initializations.

```
class Sample(Int x) {
  Int more_than_x = 5;
  { // we have two fields: x and more_than_x
    more_than_x = more_than_x + x;
  }
}
```

Active objects have their own behavior, which starts upon object creation and is specified using the special `run()` method.

```
class Active(OtherObject o) {
  Unit run() {
    o!someMethod();
    this!run(); // A self−calling run method loops forever
  }
}
```

Multitasking and Asynchronous Communication. Objects in ABS communicate via asynchronous method invocation. Each method call generates a new process in the target object, and the calling process in the caller object continues running until the result is needed. This means that processes belong to their object and never leave its scope. Calling a method produces a future variable [16,22] which is used to synchronize with the method call and get its result.

```
  Fut<Item> fi = chest!retrieve(); // asynchronous method call
  skip; // do some calculations ...
  await fi?; // suspend until call finished
  Item i = fi.get; // read value of future
```

Synchronization between processes in an object is via cooperative scheduling: no process can preempt another process. This makes it much easier to write correct programs because potential race conditions become textually apparent. For example, the method `addToQueue` is safe in ABS, since the first and second statement always execute together:

```
class IntQueue {
  List<Int> queue = Nil;
  Int counter = 0; // always contains the length of 'queue'
  Unit addToQueue(Int item) {
    counter = counter + 1;
    queue = Cons(item, queue);
  }
...
```

The **suspend** statement releases control unconditionally; the **await** *g* statement suspends the current process until the guard *g* becomes true, where *g* can be an expression over the object state, or waiting for a future. While a processes is suspended, other processes in an object may execute. At most one process can be executing in each object at any time.

Synchronous Communication. The semantics of synchronous method calls can be obtained in ABS via asynchronous method calls followed by an immediate **get** operation on the future, which *blocks* the whole object until the method call terminates. Since this is a common pattern, there is a shorthand syntax for synchronous method calls.

```
  Item i = chest.retrieve();
  // this is equivalent to:
  Fut<Item> fi = chest!retrieve();
  Item i = fi.get;
```

Initializing the Model. The dynamic behavior of models is specified using a *main block*, which is functionally equivalent to, e.g., the main function in a C program. An ABS model without a main block is syntactically valid but cannot be executed. The main block itself contains variable declarations, object instantiations, and method calls.

```
{
  // We create a chest and put in a value.
  Chest c = new LossyChest();
  Fut<Unit> f = c!putIn(Something(5));
  await f?;
}
```

Modeling Time. One characteristic of a modeling language, as opposed to a programming language, is that properties of systems are *specified* in the model instead of *measured* on the deployed system. In ABS, the progress of time can be specified, in order to associate timing properties with methods before they have been fully designed.

The function **now**`()` returns the current global time, which is modeled as a simple discrete clock [17]. The return value of **now**`()` can be used to, e.g., compare time values or monitor a system's response time. Locally, we propose to explicitly specify the progress of time by means of two statements. A process that *pauses* for some amount of time is modeled by means of the statement **await** `duration(`*min*`,`*max*`)`. In this case, the process is suspended for at least the specified amount of time, and other processes may execute. In contrast, a computation in an object which lasts for a certain amount of

time is specified by the statement `duration`(*min*, *max*). In this case the process remains active and no other process can be scheduled. Both of these statements specify time as an interval between the *min* and the *max* delay of the associated statement.

```
Time t = now(); // record current time
await duration(3, 5); // suspend process for specified duration
duration(3, 5); // block entire object for specified duration
Bool x = now() > t + 3; // true if some other process was scheduled
```

3 Deployment Components with Parametric Resources

A *deployment component* [2, 17, 18] is a resource-restricted execution context which allows us to specify and compare different execution environments for concurrent ABS models. Deployment components restrict the inherent concurrency of objects in ABS by mapping the logical concurrency to a model which includes a parametric number of available resources. These resources are shared between the component's objects.

Resource-restricted deployment components are integrated in ABS as follows. Resources are modeled by a data type `Resource` which extends the natural numbers with an "unlimited resource" ω. Resource usage is captured by resource addition and subtraction, where $\omega + n = \omega$ and $\omega - n = \omega$ (for natural numbers n). Let variables x of type `Component` refer to deployment components and allow deployment components to be statically created by the statement `x=component(`r`)`, which allocates a given quantity r of resources to the component `x`. The execution inside a deployment component is restricted by the number of available resources in the component; thus the execution in an object may need to wait for resources to become available. The availability of resources depends on the resources initially allocated to a deployment component and on the *cost model* $\mathscr{M}$ which expresses how resources become available when time advances.

All objects belong to a deployment component, so the number of objects residing on a component may grow dynamically with object creation. The ABS syntax for object creation is extended with an optional clause to specify the targeted deployment component in the expression `new C(`e`) @ x`; here, the new `C` object will reside in the component `x`.

```
Component comp = component(50); // create a deployment component
Server s = new Server() @ comp; // create an object in the deployment component
```

Objects generated without an `@`-clause reside in the same component as their parent object. The behavior of an ABS model which does not statically declare additional deployment components can be captured by a root deployment component with ω resources.

In order to give the modeler explicit control over the consumption of resources, the standard statements of ABS have zero cost and the usage of abstract resources is modeled by means of a new statement `cost(`e`)`, where e is an expression over the state variables of the object and the local variables of the method. If e evaluates to n in the current state, the execution of `cost(`e`)` in this state can only happen if at least n resources are available, and the execution of the statement consumes n resources from the deployment component. In a deployment component with ω resources, a cost statement can always be executed immediately. In a deployment component with less than n allocated resources, the statement can never be executed. Otherwise, the execution of the statement may need to wait until time has advanced in order for sufficient resources to be available.

In general, the usage of resources for objects in a deployment component depends on a specific cost model $\mathscr{M}$. During execution of a statement with cost n, the associated component consumes n resources. When time advances, resources are returned to the

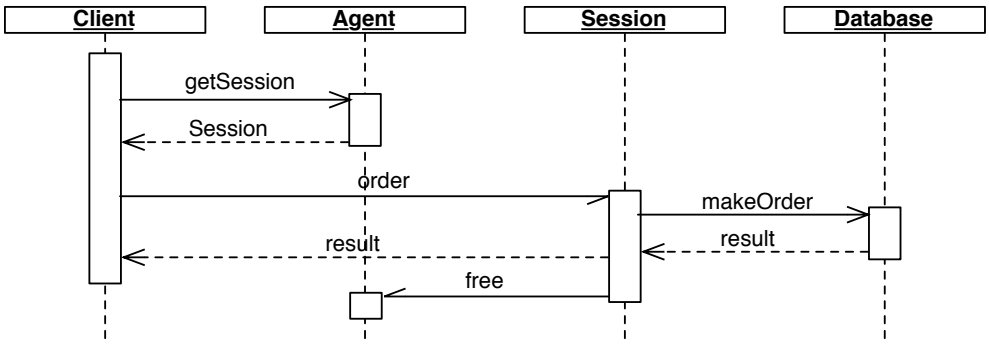

Figure 1: Example scenario

deployment components, as determined by the cost model $\mathscr{M}$. For example, for processor resources, all n resources will be returned to the deployment component when time advances. In contrast, for memory resources, the time advance corresponds to running the garbage collector, and will only return the used stack memory and a portion of the heap. Technically, for a cost transition from state t to t', the resources to be consumed by the transition are given by a function $cost_{\mathscr{M}}(t,t')$ and the resources to be returned by time advance are given by a function $free_{\mathscr{M}}(t,t')$. (Note that n is given by t, and that $cost_{\mathscr{M}}(t,t') \geq n$.) The time advance will return to the deployment component the accumulated number of resources determined by the sum of $free_{\mathscr{M}}(t,t')$ for all cost transitions in that component since the last time advance. The cost model $\mathscr{M}$ is determined by the exact definitions of $cost_{\mathscr{M}}(t,t')$ and $free_{\mathscr{M}}(t,t')$. For example, for processor resources, all resources are available in each time step so $cost_{\mathscr{M}}(t,t') = free_{\mathscr{M}}(t,t')$. For memory resources, $free_{\mathscr{M}}(t,t') = cost_{\mathscr{M}}(t,t') + size(t') - size(t)$ where $size(t)$ is a function which returns the size of the heap for state t.

4 Tool Support

The ABS language has been implemented on multiple backends, using a common parsing and compilation infrastructure. The Java backend for ABS creates Java code that is then compiled and executed on the JVM (Java Virtual Machine), with native (Java-based) and ABS-specific debugging and animation support. The formal semantics of ABS is defined in rewriting logic, which is executable on the Maude tool [8]. A code generator creates Maude terms corresponding to ABS classes and functions, which can be executed by directly using the formal semantics of ABS [16]. Our extension with cost and deployment components is currently implemented as a prototype extension of the Maude backend.

Code editing and compilation support is provided both for the Eclipse IDE and the Emacs editor. The Eclipse plugin also offers sequence chart generation and precise replays of simulation runs. Additionally, a unit testing framework and Maven-based compilation and packaging of ABS models are implemented.

5 Example: A Distributed Shopping Service

We show the design of a simple model of a web shop with a given time budget (the expected response time) for each interaction. We envisage the following main sequence of events in a client session, which is shown in Figure 1. Clients connect to the shop by calling the `getSession` method of an `Agent` object, which hands out `Session` objects

```
interface Agent { Session getSession(); Unit free(Session session);}
interface Session { Bool order(); }
interface Database { Bool makeOrder(); }

class DatabaseImp(Nat min, Nat max) implements Database {
  Bool makeOrder () {
    cost((min+max)/2);
    Time t=now();
    await duration(min, max);
    return now() <= t + max;
  }
}
class AgentImp(Database db) implements Agent {
  Set<Session> available = EmptySet;
  Session getSession() {
    cost(2);
    if isempty(available) { return new SessionImp(this, db); }
    else { session=choose(available);
      available=remove(session,available);return session;}
  }
  Unit free(Session session){
    cost(1); available=add(available,session);
  }
}
class SessionImp(Agent agent, Database db) implements Session {
  Bool order() {
    cost(2);
    return db.makeOrder();
    agent.free(this);
  }
}
```

Figure 2: A web shop model with costs in ABS

from a dynamically growing pool. Clients call the `order` method of their `Session` instance, which in turn calls the `makeOrder` method of a `Database` object that is shared across all sessions. The `order` call returns `True` if the order was completed within the specified time budget. After completing the order, the session object is returned to the agent's pool. This scenario models the architecture and control flow of a database-backed website, while abstracting from many details (load-balancing thread pools, data model, sessions spanning multiple requests, etc.), which can be added to the model if needed.

Modeling timing behavior. To model the distribution of the overall time budget among the various stages, a **cost** statement (consuming processor resources in the chosen cost model) is added at the beginning of each method. When refining the model, these costs can be adapted in accordance with a more fine-grained control flow for the implementation. Figure 2 gives a snapshot of this process in which the control flow of the methods has been designed but the cost is still specified for the whole method. The next step for modeling the timing behavior of, e.g., the method `getSession` is to move **cost** statements into the two branches of the **if** statement in the method body.

In the implementation of the `Database` class, an order takes a minimum amount of time, and should be completed within a maximum amount of time. The timing behavior of the database is configurable via the class parameters `min` and `max`.. Note that a `Database` object executed in a component with unlimited resources, will complete all orders in the

```
class SyncClient(Agent a,Nat c) {
  Unit run() {
    Time t = now(); Session s = a.getsession();
    Bool result = s.order(); await now() >= t + c; !run();
  }
}
class PeriodicClient(Agent a,Nat c) {
  Unit run() {
    Time t = now(); Session s = a.getsession();
    Fut(Bool) rc= s!order(); await now() >= t + c;
    !run(); await rc?; Bool r = rc.get;
  }
}
{ // Main block:
  Component shop = component(10);
  Database db = new DatabaseImp(5, 10) @ shop;
  Agent a = new AgentImp(db) @ shop;
  new PeriodicClient(a, 5);
}
```

Figure 3: Deployment environment and client models for the web shop example.

minimum amount of time, just as expected. In the `Agent` class, the attribute `available` stores a set of `Session` objects. (ABS has a datatype for sets, with operations `isempty` to check for the empty set, denoted `EmptySet`, `choose` to select an element of a non-empty set, and `remove` and `add` to remove or add an element to a set.) When a customer requests a `Session`, the `Agent` takes a session from the available sessions if possible, otherwise it creates a new session. The method `free` returns a session to the `available` sessions of the `Agent`, and is called by the session itself upon completion of an order.

Figure 3 shows how the web shop may be deployed: a *deployment component* `shop` is declared with 10 resources available for objects executing inside `shop`. The initial system state is given by the main block, which creates a single database, with 5 and 10 as its minimum and maximum time for orders, an `Agent` instance, and (in this example) one client outside of `shop`. The classes `SyncClient` and `PeriodicClient` model customers of the shop. `PeriodicClient` initiates a session and periodically calls `order` every `c` time intervals; `SyncClient` sends an `order` message `c` time intervals after the last call returned and hence, does not flood an overloaded shop like `PeriodicClient` does.

Figure 4 illustrates the kind of results possible to obtain after running different simulations with varying number and behavior of clients and available resources; i.e., from 10 to 50 synchronous clients and 10 to 50 available resources on the `shop` deployment component. For the synchronous client, starting with 20 clients, the number of requests goes up linearly with the number of resources, indicating that the system is running at full capacity. Moreover, the number of successful requests decreases somewhat with increasing clients since communication costs also increase. For the periodic case, the system gets overloaded much more quickly since clients will have several pending requests; hence, only 2 to 10 periodic clients were simulated. It can be seen that the system becomes completely unresponsive quickly when flooded with requests.

Testing Timed Observable Behavior. In software testing, a formal model can be used both for test case generation and as a test oracle to judge test outcomes. For example, test case generation from formal models of communication protocols can ensure that all

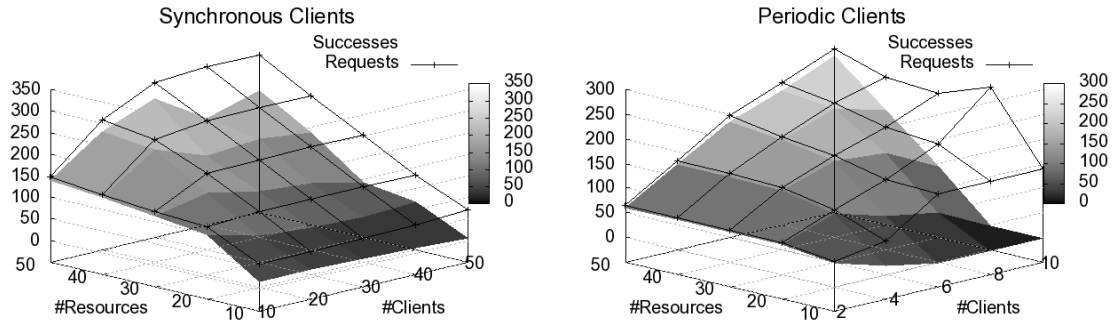

Figure 4: Number of total and successful requests, depending on the number of clients and resources, for synchronous (left) and periodic (right) clients.

possible sequences of interactions specified by the protocol are actually exercised while testing a real system. Using formal models for testing is most widely used in functionality testing (as opposed to e.g. load testing, where stability and timing performance of the system under test is evaluated), but the approaches from that area are applicable to formally specifying and testing timing behavior of software systems as well [14].

Using ABS, we can model and investigate the effects of specific deployment component configurations on the timing behavior of timed software models in the following way. The *test purpose* in this scenario would be to reach a conclusion on whether redeployment on a different configuration leads to an observable difference in timing behavior. Both *model* and *system under test* are ABS models of the same system, but running under different deployment configurations. In our example, the client object(s) model the expected usage scenario; results about test success or failure are relative to the expected usage. As *conformance relation* trace equivalence can be used – this simple relation is sufficient since model and system under test have the same internal structure, hence there is no need to test for input enabledness, invalid responses etc. Traces are sequences of communication events, (i.e., method invocations and responses) annotated with the time of occurrence, which can be recorded on both the model and the system under test and then compared after the fact (off-line testing).

If the traces differ *in the timing* for some test scenario consisting of input behavior plus resource configuration, the resource allocation influences system timing behavior in an observable way. If the sequence of operations itself is different, the functional aspects of the model are influenced by the resource allocation as well.

6 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously, is increasingly getting attention due to its intuitive and compositional nature [3, 12, 22]. This compositionality allows concurrent objects to be naturally distributed on different locations, because only the local state of a concurrent object is needed to execute its methods. In previous work [2, 17, 18], the authors have introduced *deployment components* as a modeling concept which captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model's behavior for different assumptions about the available resources. In that work, the cost of execution was bound to a specific resource and directly fixed in the language semantics. In contrast, this paper generalizes that approach by

proposing the specification of resource costs as part of the software development process, supported by explicit user-defined cost statements expressed in terms of the local state and the input parameters to methods. This way, the cost of execution in the model may be adapted by the modeler to a specific cost scenario. Our extension to ABS allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of allocated resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system.

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [9]. A survey of model-based performance analysis techniques is given in [4]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [5, 10]), but also to the schedulability of processes in concurrent objects [15]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but associates deadlines with method calls with abstract duration statements.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [20] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [4]. Closer to our work is M. Verhoef's extension of VDM++ for embedded real-time systems [21], in which architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software. Verhoef's approach is also based on abstract executable modeling, but the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller decided synchronization.

Another interesting line of research is static cost analysis for object-oriented programs. Most tools for cost analysis only consider sequential programs. Our approach, in which the modeler determines how to specify the cost in the cost statements, could be supported by a cost analysis tool such as COSTA [1]). In collaboration with Albert et al., we have previously used this approach for memory analysis of ABS models [2]. However, the generalization of that work for general, user-defined resources and its integration into the software development process remains future work.

7 Conclusion

Software modeling and analysis usually abstracts from low-level deployment aspects. As software is increasingly being developed for different architectures, there is a need to model and to reason about deployment choices and about how a targeted architecture affects the behavior of a software system. The resources which are made available to a software component influence the quality of service offered by the component even if the functionality is unchanged, for example with respect to response time. Modeling languages today do not meet this need in a satisfactory way. In order to specify and analyze the effect of deployment choices early in the software development process, these

choices must be expressible at the abstraction level of the modeling language.

As a step in this direction, we propose to represent certain low-level deployment aspects in high-level modeling languages in terms of resource allocation and usage. In order to integrate resource usage in a natural way, it is essential that the model reflects the control flow of the system. We argue that abstract executable modeling languages offer the best abstraction level for integrating deployment aspects. Our approach is to introduce deployment components, which act as resource-restricted execution contexts for a group of concurrent objects, and are parametric in the amount of resources they make available to their objects. This makes it possible to analyze the behavior of a model ranging over the resources available in different deployment scenarios.

In this paper, we have given an informal overview over our approach based on deployment components. We have also proposed to integrate resource specifications in the software engineering process, so performance of a model can be analyzed at an early stage of system development and performance requirements can influence the design of the system's fine-grained control flow. This is done by introducing explicit cost statements in the executable modeling language that are refined as the low-level control flow is specified. We have illustrated this process in terms of a web shop example, and showed how our simulation tool was used to validate performance in terms of soft real-time requirements as the resources of a model's deployment components vary.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46:161–203, 2011.

[2] 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 *Proc. Formal Methods (FM 2011)*, *LNCS* 6664, pages 353–368. Springer, June 2011.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[5] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. 3rd Intl. Conf. on Embedded Software (EMSOFT'03)*, *LNCS* 2855, pages 117–133. Springer, 2003.

[6] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proc. 4th Intl. Symp. on Formal Methods for Components and Objects (FMCO'04)*, *LNCS* 4111, pages 342–363. Springer, 2005.

[7] D. Clarke, *et al.* Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, *LNCS* 6659, pages 417–457. Springer, 2011.

[8] M. Clavel, *et al. All About Maude - A High-Performance Logical Framework*, *LNCS* 4350. Springer, 2007.

[9] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. ICSE*, pages 111–121. IEEE, 2009.

[10] E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[11] N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, Sept. 1992.

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[13] I. Hayes and C. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.

[14] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *Proc. Formal Methods and Testing*, *LNCS* 4949, pages 77–117. Springer, 2008.

[15] M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*, 78(5):402–416, 2009.

[16] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. Formal Methods for Components and Objects (FMCO 2010)*. To appear in *LNCS*. Springer, 2011.

[17] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. Formal Engineering Methods (ICFEM'10)*, *LNCS* 6447, pages 646–661. Springer, Nov. 2010.

[18] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Proc. Formal Verification of Object-Oriented Software (FoVeOOS'10)*, *LNCS* 6528, pages 46–60. Springer, 2011.

[19] J. Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, 2007.

[20] D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and System Modeling*, 6(2):163–184, 2007.

[21] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. Formal Methods (FM'06)*, *LNCS* 4085, pages 147–162. Springer, 2006.

[22] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA*, pages 439–453. ACM Press, 2005.

[23] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.