Simulating Concurrent Behaviors with Worst-Case Cost Bounds *

Elvira Albert[1], Samir Genaim[1], Miguel Gómez-Zamalloa[1],
Einar Broch Johnsen[2], Rudolf Schlatte[2], and S. Lizeth Tapia Tarifa[2]

[1] DSIC, Complutense University of Madrid, Spain
{elvira,samir.genaim,mzamalloa}@fdi.ucm.es

[2] Department of Informatics, University of Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

Abstract. Modern software systems are increasingly being developed for deployment on a range of architectures. For this purpose, it is interesting to capture aspects of low-level deployment concerns in high-level modeling languages. In this paper, an executable object-oriented modeling language is extended with resource-restricted deployment components. To analyze model behavior a formal methodology is proposed to assess resource consumption, which balances the scalability of the method and the reliability of the obtained results. The approach applies to a general notion of resource, including traditional cost measures (e.g., time, memory) as well as concurrency-related measures (e.g., requests to a server, spawned tasks). The main idea of our approach is to combine reliable (but expensive) worst-case cost analysis of statically predictable parts of the model with fast (but inherently incomplete) simulations of the concurrent aspects in order to avoid the state-space explosion. The approach is illustrated by the analysis of memory consumption.

1 Introduction

Software systems today are increasingly being developed to be highly configurable, not only with respect to the functionality provided by a specific instance of the system but also with respect to the targeted deployment architecture. An example of a development method is software product line engineering [20]. In order to capture and analyze the intended deployment variability of such software, formal models need to express and range over different *deployment scenarios*. For this purpose, it is interesting to reflect aspects of low-level deployment in high-level modeling languages. As our first contribution, in this paper, we propose a notion of *resource-restricted* deployment component for an executable modeling language based on *concurrent objects* [8, 11, 14, 21, 24]. The main idea of resource-restricted deployment components is that they are parametric in the amount of resources they make available to their concurrently executing objects.

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

This way, different deployment scenarios can be conveniently expressed at the modeling level and a model may be analyzed for a range of deployment scenarios.

As our main contribution, we develop a novel approach for estimating the *resource consumption* of this kind of resource-constrained concurrent executions which is reasonably reliable and scalable. Resource consumption is in this sense a way of understanding and debugging the model of the deployment components. Our work is based on a general notion of *resource*, which can be any function that associates a cost unit to the program statements. Traditional resources are execution steps, time and memory, but one may also consider more concurrency-related resources like the number of tasks spawned along the execution, the number of requests to a server, etc.

The two main approaches to estimating resource consumption of a program execution are *static cost analysis* and *dynamic simulation* (or monitoring). Efficient simulation techniques can analyze model behavior in different deployment scenarios, but simulations are carried out for particular input data. Hence, they cannot guarantee the correctness of the model. Due to the non-determinism of concurrent execution and the choice of inputs, possible errors may go undetected in a simulation. Static cost (or resource usage) analysis aims at automatically inferring a program's resource consumption *statically*, i.e., without running the program. Such analysis must consider all possible execution paths and ensures soundness, i.e., it guarantees that the program never exceeds the inferred resource consumption for any input data. While cost analysis for sequential languages exists, the problem has not yet been studied in the concurrent setting, partly due to the inherent complexity of concurrency: the number of possible execution paths can be extremely large and the resulting outcome non-deterministic. Statically analyzing the concurrent behaviors of our resource-restricted models requires a full state space exploration and quickly becomes unrealistic.

In this paper, we propose to combine simulations with static techniques for cost analysis, which allows classes of input values to be covered by a single simulation. The main idea is to apply cost analysis to the sequential computations while simulation handles the concurrent system behavior. Our method is developed for an abstract behavioral specification language *ABS*, simplifying Creol [11,14], which contains a functional level where computations are sequential and an concurrent object-oriented level based on concurrent objects. This separation allows a concise and clean formalization of our technique. The combination of simulation and static analysis, as proposed in this paper, suggests a middle way between full state space exploration and simulating single paths, which gives interesting insights into the behavior of concurrent systems.

Paper organization. Section 2 describes the syntax and semantics of the ABS modeling language and introduces our running example. Section 3 discusses the worst-case cost bounds calculation of the functional parts of ABS. Section 4 introduces deployment components, which model resource-containing runtime entities, and Section 5 presents the results of applying the technique to the running example. Finally, Section 6 discusses related work and Section 7 concludes the paper with a discussion of extensions of our presented technique.

Syntactic categories.	*Definitions.*
I in Interface type	$Dd ::= \mathbf{data}\ D = Cons;$
D in Data type	$Cons ::= Co[(\overline{T})] \mid (Cons \mid Cons)$
x in Variable	$F ::= \mathbf{def}\ T\ fn(\overline{T\ x}) = e;$
e in Expression	$T ::= I \mid D$
b in Bool Expression	$e ::= b \mid x \mid t \mid \mathbf{this} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \mathbf{case}\ e\ \{\overline{br}\}$
t in Ground Term	$t ::= Co[(\overline{t})] \mid \mathbf{null}$
br in Branch	$br ::= p \Rightarrow e;$
p in Pattern	$p ::= \_ \mid x \mid t \mid Co[(\overline{p})]$

Fig. 1. ABS syntax for the functional level. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [] optional elements. Boolean expressions b include comparison by equality, greater- and less-than operators.

2 A Language for Distributed Concurrent Objects

Our method is presented for *ABS*, an abstract behavioral specification language for distributed concurrent objects (simplifying Creol [11, 14] by excluding, e.g., class inheritance and dynamic class upgrades). Characteristic features of ABS are that: (1) it allows abstracting from implementation details while remaining executable; i.e., a *functional sub-language* over abstract data types is used to specify internal, sequential computations; and (2) it provides *flexible concurrency and synchronization mechanisms* by means of asynchronous method calls, release points in method definitions, and cooperative scheduling of method activations.

Intuitively, concurrent ABS objects have dedicated processors and live in a distributed environment with asynchronous and unordered communication. All communication is between named objects, typed by interfaces, by means of asynchronous method calls. (There is no remote field access.) Calls are asynchronous as the caller may decide at runtime when to synchronize with the reply from a call. Method calls may be seen as triggers of concurrent activity, spawning new activities (so-called *processes*) in the called object. Active behavior, triggered by an optional *run* method, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active* and the others are *suspended* in a process pool. Process scheduling is non-deterministic, but controlled by *processor release points* in a cooperative way.

An ABS *model* defines interfaces, classes, datatypes, and functions, and has a `main` method to configure the initial state. Objects are dynamically created instances of classes; their declared attributes are initialized to arbitrary type-correct values, but may be redefined in an optional method *init*. This paper assumes that models are well-typed, so method binding is guaranteed to succeed.

The functional level of ABS defines data types and functions, as shown in Fig. 1. In data type declarations Dd, a data type D has at least one constructor $Cons$, which has a name Co and a list of types T for its arguments. Function declarations F consist of a return type T, a function name fn, a list of variable declarations $\overline{x}$ of types $\overline{T}$, and an expression e. *Expressions* e include

Syntactic categories.	*Definitions.*
C, m in Names	$IF ::= \texttt{\textbf{interface}}\ I\ \{\ \overline{Sg}\ \}$
g in Guard	$CL ::= \texttt{\textbf{class}}\ C\ [(\overline{T\ x})]\ [\texttt{\textbf{implements}}\ \overline{I}]\ \{\ \overline{T\ x};\ \overline{M}\}$
s in Statement	$Sg ::= T\ m\ (\overline{T\ x})$
	$M ::= Sg\ \{\ \overline{T\ x};\ s\ \}$
	$g ::= b \mid x? \mid g \wedge g \mid g \vee g$
	$s ::= s; s \mid x := rhs \mid \texttt{\textbf{release}} \mid \texttt{\textbf{await}}\ g \mid \texttt{\textbf{return}}\ e$
	$\mid \texttt{\textbf{if}}\ b\ \texttt{\textbf{then}}\ \{\ s\ \}\ [\texttt{\textbf{else}}\ \{\ s\ \}] \mid \texttt{\textbf{while}}\ b\ \{\ s\ \} \mid \texttt{\textbf{skip}}$
	$rhs ::= e \mid \texttt{\textbf{new}}\ C[(\overline{e})] \mid [e]!m(\overline{e}) \mid x.\texttt{\textbf{get}}$

Fig. 2. ABS syntax for the concurrent object level.

Boolean expressions b, variables x, (ground) terms t, the (read-only) variable **this** which refers to the object's identifier, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$. Ground terms t are constructors applied to ground terms $Co(\overline{t})$, and **null**. Case expressions have a list of branches $p \Rightarrow e$, where p is a pattern. The branches are evaluated in the listed order. Patterns include wild cards _, variables x, terms t, and constructor patterns $Co(\overline{p})$. Remark that expressions may refer to object references.

Example 1. Consider a polymorphic data type for sets and a function `in` which checks if `e` is an a member of the set `ss`.

```
data Set<A> = EmptySet | Insert(A, Set<A>);
def Bool in<A>(Set<A> ss, A e) =
  case ss {EmptySet => False ;
           Insert(e, _) => True;
           Insert(_, xs) => in(xs, e); };
```

The concurrent object level of ABS is given in Fig. 2. Here, an interface IF has a name I and method signatures Sg. A class implements a list of interfaces, specifying types for its instances; a class CL has a name C, interfaces $\overline{I}$, class parameters and state variables x of type T, and methods M (The *attributes* of the class are both its parameters and state variables). A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, a list of local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access attributes of the current class, locally defined variables, and the method's formal parameters.

Right hand side expressions rhs include object creation **new** $C(\overline{e})$, method calls, and (pure) expressions e. Statements are standard for assignment $x := rhs$, sequential composition $s_1; s_2$, and **skip**, **if**, **while**, and **return** constructs. **release** unconditionally releases the processor, suspending the active process. In **await** g, the guard g controls processor release and consists of Boolean conditions b and return tests $x?$ (see below). If g evaluates to false, the processor is released and the process *suspended*. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled. Explicit signaling is therefore redundant. Like expressions e, guards g are side-effect free.

Communication in ABS is based on asynchronous method calls, denoted $o!m(\overline{e})$. (Local calls are written $!m(\overline{e})$.) 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 (an expression 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 external synchronization in ABS. First, a return test $x?$ evaluates to false unless the reply to the call can be retrieved. (Return tests are used in guards.) 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*, abbreviated **await** $v := o.m(\overline{e})$.

Example 2. Consider a model of a book shop where clients can order a list of `books` for delivery to a `country`. Clients connect to the shop by calling the `getSession` method of an `Agent` object. An `Agent` hands out `Session` objects from a dynamically growing pool. Clients call the `order` method of their `Session` instance, which calls the `getInfo` and `confirmOrder` methods of a `Database` object shared between the different sessions. `Session` objects return to the agent's pool after an order is completed. (The full model is available in [5].)

```
interface Agent { Session getSession(); Unit free(Session session);}
interface Session {
         OrderResult order(List<Bname> books, Cname country);}
interface Database {
         DatabaseInfo getInfo(List<Bname> books, Cname country);
         Bool confirmOrder(List<Bname> books); }
class  DatabaseImp(Map<Bname,Binfo> bDB, Map<Cname,Cinfo> cDB)
implements Database {
 DatabaseInfo getInfo(List<Bname> books, Cname country){
  Map<Bname,Binfo> bOrder:=EmptyMap; Pair<Cname,Cinfo> cDestiny;
  bOrder:=getBooks(bDB, books); cDestiny:=getCountry(cDB, country);
  return Info(bOrder, cDestiny);} ...
```

In the model, a `DatabaseImp` class stores and handles the information about the books available in the shop (in the `bDB` map) as well as information about the delivery countries (in the `cDB` map). This class has a method `getInfo`; given an order with a list of `books` and a destination `country`, the `getInfo` method extracts information about book availability from `bDB` and shipping information from `cDB` by means of function calls `getBooks(bDB, books)` and `getCountry(cDB, country)` The result from the method call has type `DatabaseInfo`, with a constructor of the form: `Info(bOrder, cDestiny)`.

2.1 Operational Semantics

The operational semantics of ABS is presented as a transition system in an SOS style [19]. Rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [18]). A run is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the application of transition rules in a run.

Configurations cn are sets of objects, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by ε. These configurations live inside curly brackets; in the term $\{cn\}$, cn captures the *entire* configuration. An *object* is a term $ob(o, C, a, p, q)$ where o is the object's identifier and C its class, a an attribute mapping representing the object's fields, p an *active process*, and q a *pool of suspended processes*. A process p consists of a mapping l of local variable bindings and a list s of statements, denoted by $\{l|s\}$ when convenient. In an *invocation message* $invoc(o, f, m, \overline{v})$, o is the callee, f the future to which the call's result is returned, m the method name, and $\overline{v}$ the call's actual parameter values. A *future* $fut(f, v)$ has a identifier f and a reply value v (which is $\bot$ when the future's reply value has not been received). Values are object and future identifiers, Boolean expressions, and null (as well as expressions in the functional language). For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables.

Evaluating Expressions. Denote by $\sigma(x)$ the value bound to x in a mapping σ and by $\sigma_1 \circ \sigma_2$ the composition of mappings σ_1 and σ_2. Given a substitution σ and a configuration cn, denote by $[\![e]\!]_\sigma^{cn}$ a confluent and terminating reduction system which reduces expressions e to data values. Let $[\![x?]\!]_\sigma^{cn} = \text{true}$ if $[\![x]\!]_\sigma^{cn} = f$ and $fut(f, v) \in cn$ for some value $v \neq \bot$, otherwise $[\![x?]\!]_\sigma^{cn} = \text{false}$. The remaining cases are fairly straightforward, looking up values for declared variables in σ. For brevity, we omit the reduction system for the functional level of ABS (for details, see [5]) and simply denote by $[\![e]\!]_\sigma^{\varepsilon}$ the evaluation of a guard or expression e in the context of a substitution σ and a state configuration cn (the state configuration is needed to evaluate future variables). The reduction of an expression always happens in the context of a given process, object state, and configuration. Thus, $\sigma = a \circ l$ (the composition of the fields a and the local variable bindings l), and cn the current configuration of the system (ignoring the object itself).

Transition Rules. Transition rules of the operational semantics transform state configurations into new configurations, and are given in Fig. 3. We assume given functions $bind(o, f, m, \overline{v}, C)$ which returns a process resulting from the method activation of m in a class C with actual parameters $\overline{v}$, callee o and associated future f; $init(C)$ which returns a process initializing instances of class C; and $atts(C, \overline{v}, o, n)$ which returns the initial state of an instance of class C with class parameters $\overline{v}$, identity o, and deployment component n. The predicate $fresh(n)$ asserts that a name n is globally unique (where n may be an identifier for an object or a future). Let *idle* denote any process $\{l|s\}$ where s is an empty statement list. Finally, we define different assignment rules for side effect free expressions (*assign1* and *assign2*), object creation (*new-object*), method calls (*async-call*), and future dereferencing (*read-fut*). Rule *skip* consumes a **`skip`** in the active process. Here and in the sequel, the variable s will match any (possibly empty) statement list. Rules *assign1* and *assign2* assign the value of expression

$$\textsc{(skip)}\quad ob(o,C,a,\{l|\mathrm{skip};s\},q) \rightarrow ob(o,C,a,\{l|s\},q)$$

$$\textsc{(release)}\quad ob(o,C,a,\{l|\mathrm{release};s\},q) \rightarrow ob(o,C,a,\mathrm{idle},\mathrm{enqueue}(\{l|s\},q))$$

$$\textsc{(activate)}\quad \frac{p = \mathrm{select}(q,a,cn)}{\{ob(o,C,a,\mathrm{idle},q)\ cn\} \rightarrow \{ob(o,C,a,p,q\backslash p)\ cn\}}$$

$$\textsc{(Async-Call)}\quad \frac{o' = [\![e]\!]^{\varepsilon}_{(a\circ l)} \quad \overline{v} = [\![\overline{e}]\!]^{\varepsilon}_{(a\circ l)} \quad \mathrm{fresh}(f)}{ob(o,C,a,\{l|x := e!m(\overline{e});s\},q) \rightarrow ob(o,C,a,\{l|x := f;s\},q)\ invoc(o',f,m,\overline{v})\ fut(f,\bot)}$$

$$\textsc{(New-Object)}\quad \frac{\mathrm{fresh}(o')\ p = \mathrm{init}(B)\ a' = \mathrm{atts}(B,[\![\overline{e}]\!]^{\varepsilon}_{a\circ l},o',n)}{ob(o,C,a,\{l|x := \mathrm{new}\ B(\overline{e});s\},q) \rightarrow ob(o,C,a,\{l|x := o';s\},q)\ ob(o',B,a',p,\emptyset)}$$

$$\textsc{(return)}\quad \frac{v = [\![e]\!]^{\varepsilon}_{(a\circ l)} \quad l(\mathrm{destiny}) = f}{ob(o,C,a,\{l|\mathrm{return}\ e;s\},q)\ fut(f,\bot) \rightarrow ob(o,C,a,\{l|s\},q)\ fut(f,v)}$$

$$\textsc{(Read-Fut)}\quad \frac{v \neq \bot \quad f = [\![e]\!]^{\varepsilon}_{(a\circ l)}}{ob(o,C,a,\{l|x := e.\mathrm{get};s\},q)\ fut(f,v) \rightarrow ob(o,C,a,\{l|x := v;s\},q)\ fut(f,v)}$$

$$\textsc{(Bind-Mtd)}\quad \frac{p' = \mathrm{bind}(o,f,m,\overline{v},C)}{ob(o,C,a,p,q)\ invoc(o,f,m,\overline{v}) \rightarrow ob(o,C,a,p,\mathrm{enqueue}(p',q))}$$

$$\textsc{(assign1)}\quad \frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]^{\varepsilon}_{(a\circ l)}}{ob(o,C,a,\{l|x := e;s\},q) \rightarrow ob(o,C,a,\{l[x \mapsto v]|s\},q)}$$

$$\textsc{(assign2)}\quad \frac{x \in \mathrm{dom}(a) \quad v = [\![e]\!]^{\varepsilon}_{(a\circ l)}}{ob(o,C,a,\{l|x := e;s\},q) \rightarrow ob(o,C,a[x \mapsto v],\{l|s\},q)}$$

$$\textsc{(await1)}\quad \frac{\neg[\![g]\!]^{cn}_{(a\circ l)}}{\{ob(o,C,a,\{l|\mathrm{await}\ g;s\},q)\ cn\} \rightarrow \{ob(o,C,a,\{l|\mathrm{release};\mathrm{await}\ g;s\},q)\ cn\}}$$

$$\textsc{(await2)}\quad \frac{[\![g]\!]^{cn}_{(a\circ l)}}{\{ob(o,C,a,\{l|\mathrm{await}\ g;s\},q)\ cn\} \rightarrow \{ob(o,C,a,\{l|s\},q)\ cn\}}$$

Fig. 3. ABS Semantics

e to a variable x in the local variables l or in the fields a, respectively. (We omit the standard rules for if-then-else and while).

Process Suspension and Activation. Three operations are used to manipulate a process pool q: enqueue(p, q) adds a process p to q, $q \setminus p$ removes p from q, and select(q, a, cn, t) selects a process from q (which is `idle` if q is empty or no process is *ready* [14]). The actual definitions are left undefined; different definitions correspond to different process scheduling policies. Let $\emptyset$ denote the empty pool. Rule *release* suspends the active process to the pool, leaving the active process idle. Rule *await1* consumes the await statement if the guard evaluates to true in the current state of the object, rule *await2* adds a release statement in order to suspend the process if the guard evaluates to false. Rule *activate* selects a process from the pool for execution if this process is *ready* to execute, i.e., if it would not directly be resuspended or block the processor [14].

Communication and Object Creation. Rule *async-call* sends an invocation message to o' with the unique identity f (by the condition fresh(f)) of a new future, the method name m, and actual parameters $\overline{v}$. Note that the return value of the new future f is undefined (i.e., $\bot$). Rule *bind-mtd* consumes an invocation method and places the process corresponding to the method activation in the process pool of the callee. Note that a reserved local variable 'destiny' is used to store the identity of the future associated with the call. Rule *return* places the return value into the call's associated future. Rule *read-fut* dereferences the future f in the case where $v \neq \bot$. Note that if this attribute is $\bot$ the reduction in this object is *blocked*. Finally, *new-object* creates a new object with a unique identifier o'. The object's fields are given default values by $\text{atts}(B, \overline{v}, o', n)$, extended with the actual values $\overline{v}$ for the class parameters and o' for this. In order to instantiate the remaining attributes, the process p is loaded (we assume that this process reduces to idle if $\text{init}(B)$ is unspecified in the class definition, and that it asynchronously calls `run` if the latter is specified).

3 Worst-Case Cost Bounds

The goal of this section is to infer *worst-case upper bounds* (UBs) from the (sequential) functions in our sub-language. This problem has been intensively studied since the seminal paper on cost analysis [23]. Thus, instead of a formal development, we illustrate the main steps of the analysis on the running example.

Size of terms. The cost of a function that traverses a term t usually depends on the *size* of t, and not on the concrete data structure to which t is bound. For instance, the cost of executing *dom*(*map*) (which returns the domain of a map) depends on the size of *map* (the number of elements). Therefore, in order to infer worst-case UBs, we first need to define the meaning of *size of a term*. This is done by using *norms* [7]. A norm is a function that maps terms to their size. For instance, the *term-size* norm calculates the number of type constructors in a given term, and is defined as $|Co(t_1, \ldots, t_n)|_{ts} = 1 + \Sigma_{i=1}^{n} |t_i|_{ts}$, and, the *term-depth* norm calculates the depth of the term, and is defined as $|Co(t_1, \ldots, t_n)|_{td} = 1 + \max(|t_1|_{td}, \ldots, |t_n|_{td})$. Consider the book shop model described in Ex. 2; the database uses maps for storing information; a `Map<A, B>` has two constructors `Ins(Pair<A, B>, Map<A, B>)` and `EmptyMap` (to represent empty maps). For storing the information of a book sold by the shop, the model uses a constructor of the form `BInfo(Bquantity, Bweight, Bbackordertime)` (A more detailed description of this data type can be found in [5].). For a term:

$t =$ `Ins(Pair("b1",BInfo(5,1,2)),Ins(Pair("b2",BInfo(1,2,5)),EmptyMap))`

which can represent the database of books in the shop, we have that $|t|_{ts} = 15$ and $|t|_{td} = 5$. Note that we count strings and numbers as type constructors. Norms map a given variable x to itself in order to account for the size of the term to which x is bounded. Any norm can be used in the analysis, depending on the used data structures, w.l.o.g., we will use the term-size norm.

Size relations. The `getBooks` function (called from method `getInfo` in Ex. 2) returns a sub-database (of `booksDB`) which contains only those books in `books`:

```
def Map getBooks(Map booksDB,List books) = case books {
  Nil => EmptyMap;
  Cons(b,t) => case in(dom(booksDB),b) {
    False => getBooks(booksDB,t) ;
    True => Ins(Pair(b,lookup(booksDB,b)),getBooks(booksDB,t)); };};
```

Function `dom` returns the set of keys of the mapping provided as argument, `in` is the one of Ex. 1, and, `lookup` returns the value that corresponds to the provided key in the provided mapping. Observe that the return value of `dom` is passed on to function `in`. Since the cost of `in` is part of the total cost of `getBooks`, we need to express its cost in terms of `booksDB`. This is possible only if we know which is the relation between the returned value of `dom` and its input value `booksDB`. This *input-output* relation (or a post-condition) is a conjunction of (linear) constraints that describe a relation between the sizes of the input parameters of the function and its return value, w.r.t. the selected norm. E.g., $ret \leq map$ is a possible post-condition for function *dom*, where *map* is the size of its input parameter and *ret* is the size of the returned term. We apply existing techniques [6] to infer such relations for our functional language. In what follows, we assume that $\mathcal{I}_P$ includes a post-conditions $\langle fn(\bar{x}), \psi \rangle$ for each function, where ψ is a conjunction of (linear) constraints over $\bar{x}$ and ret.

Cost Model. Cost analysis is typically parametric on the notion of *cost model* $\mathcal{M}$, i.e., on the resource that we want to measure [2]. Informally, a cost model is a function that maps instructions to costs. Traditional cost models are: (1) *number of instructions*, which maps all instructions to 1, i.e., $\mathcal{M}(b) = 1$ for all instructions b; and (2) *memory consumption*, which can be defined as $\mathcal{M}_h(x = t) = \mathcal{M}_h(t) = mem(t)$ where $mem(Co(t_1, \ldots, t_n)) = \mathsf{Co} + \Sigma_{i=1}^{n} mem(t_i)$ and $mem(x) = 0$. For any other instruction b we let $\mathcal{M}_h(b) = 0$. The symbol Co represents the amount of memory required for constructing a term of type Co. Note that we estimate only the memory required for storing terms.

Upper bounds. In order to make the presentation simpler, we assume functions are normalized such that nested expressions are flattened using **`let`** bindings. Using this normal form, the evaluation of an expression e consists in evaluating a sequence of sub-expressions of the form $y = fn(\bar{x})$, $y = t$, $match(y, t)$, $fn(\bar{x})$, t or x. We refer to such sequence as an execution path of e. In a static setting, since variables are not assigned concrete values, and due to the use of **`case`**, an expression e might have several execution paths. We denote the set of all execution paths of e by $paths(e)$. This set can be constructed from the abstract syntax tree of e. Clearly, when estimating the cost of executing an expression e we must consider all possible execution paths. In practice, we generate a set of (recursive) equations where each equation accounts for the cost of one execution path. Then, the solver of [1] is used in order to obtained a closed-form UB.

Definition 1. *Given a function* **`def`** $T\ fn(\overline{T\ x}) = e$*, its cost relation (CR) is defined as follows: for each execution path* $p \equiv b_1, \ldots, b_n \in paths(e)$*, we define an equation* $\langle fn(\bar{x}) = \Sigma_{i=1}^{n} \mathcal{M}(b_i) + fn_{i_1}(\bar{x}_{i_1}) + \cdots + fn_{i_k}(\bar{x}_{i_k}), \wedge_{i=1}^{n} \varphi_i \rangle$ *where* $fn_{i_1}(\bar{x}_{i_1}), \ldots, fn_{i_k}(\bar{x}_{i_k})$ *are all function calls in p; and* $\varphi_i \equiv y = |t|_{ts}$ *if* $b_i \equiv y = t$*, and* $\varphi_i \equiv \psi[ret/y]$ *if* $b_i \equiv y = f(\bar{x})$ *and* $\langle f(\bar{x}), \psi \rangle \in \mathcal{I}_P$*, otherwise* $\varphi_i = true$*. The CR system of a given program the set of all CRs of its functions.*

Example 3. The following is the *CR* of `getBooks` w.r.t the cost model *mem*:

$$getBooks(a,b) = \mathsf{EmptyMap} \quad \{b=1\}$$
$$getBooks(a,b) = dom(a)+in(d,e)+getBooks(a,g) \quad \{b=1+e+g, d\leq a, d\geq 1, e\geq 1, g\geq 1\}$$
$$getBooks(a,b) = \mathsf{Pair}+\mathsf{Ins}+dom(a)+in(d,e) + lookup(a,e)+getBooks(a,g) \quad \{b=1+e+g, d\leq a, d\geq 1, e\geq 1, g\geq 1\}$$

The first equation can be read as "the memory consumption of $getBooks$ is one $EmptyMap$ constructor if the size of b is 1". Equations for functions in, $lookup$ and dom are not shown due to space limitations and have resp. constant, zero and linear memory consumptions. Solving the above CR results in the UB

$$getBooks(a,b) = \mathsf{EmptyMap}+\mathsf{nat}(\tfrac{b-1}{2})*(\mathsf{nat}(\tfrac{a-1}{4})*\mathsf{Ins}+\mathsf{EmptySet}+\max(\mathsf{True},\mathsf{False}))$$

Replacing, for example, EmptyMap, Ins, True and False by 1 results in

$$getBooks(a,b) = 1 + \mathsf{nat}(\tfrac{b-1}{2}) * (2 + \mathsf{nat}(\tfrac{a-1}{4}))$$

4 Deployment components

Deployment components make quantifiable deployment-level resources explicitly available in the modeling language. A deployment component allows the logical execution environment of concurrent objects to be mapped to a model of physical resources, by specifying an abstract execution context which is shared between a number of concurrently executing objects. The resources available to a deployment component are shared between the component's objects. An object may get and return resources from and to its deployment component. Thus, the deployment components impose a resource-restricted execution context for their concurrently executing objects, but not a communication topology as objects still communicate directly with each other independent of the components.

Resource-restricted deployment components are integrated in the modeling language as follows. Let variables x of type `Component` refer to deployment components and allow deployment components to be statically created by the statement x`:=`**`component`**`(`r`)` in the main method, which allocates a given quantity of resources r to the component x (capturing the resource constraint of x). Resources are modeled by a data type `Resource` which extends the natural numbers with an "unbounded resource" ω. Resource allocation and usage is captured by resource addition and subtraction, where $\omega + n = \omega$ and $\omega - n = \omega$.

Concurrent objects residing on components, may grow dynamically. All objects are created inside a deployment component. The syntax for object creation is extended with an optional clause to specify the targeted deployment component in the expression **`new`** C`(`$\overline{e}$`)@`x. This expresses that the new C object will reside in the component x. Objects generated without an `@` clause reside in the same component as their parent object. Thus 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.

Example 4. Consider the book shop model described in Ex. 2 instantiated inside deployment components:

```
Component c := component(200);
Database db := new DataBaseImp(...) @ c;
Agent agent := new AgentImp(db) @ c;
```

Rule	cost	free
ASSIGN1, ASSIGN2	$\text{cost}(e)$	$\|vp\| - \|v\|$
READ-FUT	$max(\text{cost}(e), \|v\|)$	0
BIND-MTD	$P + \|\overline{v}\|$	$-(P + \|\overline{v}\|)$
ASYNC-CALL	$\text{cost}(\overline{e}) + \|f\|$	0
NEW-OBJECT-CREATE	$O + P + \|\overline{v}\|$	$-(O + P + \|\overline{v}\|)$

Table 1. The non-trivial cost functions of memory-constrained ABS semantics. All identifiers are the same as in the corresponding rule of Figure 3, except vp (old value of a variable), $|v|$ (size of term v), P (size of a process), and O (size of an object).

The `Session` objects created and handed out by the `Agent` object will then be created inside `c` as well, without further changes to the model.

The *execution* inside a component d with r resources can be understood as follows. An object o residing in d may execute a transition step with cost c if

- o can execute the step in a context with unbounded resources, and
- $c \leq r$; i.e., the cost of executing the step does not exclude the transition in an execution context restricted to r resources.

After the execution of the transition step, the object may return free resources to its deployment component. Thus, for each transition rule the resources needed to apply this rule to a state t, resulting in a state t', can be characterized in terms of two functions over the state space, one computing the *cost* of the transition form t to t' and one computing the *free* resources after the transition. The allocation and return of resources for objects in a deployment component will depend on the specific cost model $\mathcal{M}$ for the considered resource, so the exact definitions of $cost_{\mathcal{M}}(t, t')$ and $free_{\mathcal{M}}(t, t')$ depend on $\mathcal{M}$.

Example 5. Table 1 shows the $cost_{\mathcal{M}}(t, t')$ and $free_{\mathcal{M}}(t, t')$ functions for the memory cost model of the ABS semantics, using the symbols of Figure 3. There are some subtle details in these functions – for example, message invocations and future variables are considered to be outside any one deployment component, so the memory required to execute the READ-FUT rule can be larger than evaluating the future variable expression e since the deployment component must have enough memory to accommodate the incoming value v. Also, object creation affects two places, so was split into two rules, similar to method invocation.

Semantics of Resource Constrained Execution. Let $\mathcal{M}$ be a cost model. The operational semantics of $\mathcal{M}$-constrained execution in deployment components is defined as a small-step operational semantics, extending the semantics of ABS given in Sec. 2.1 to resource-sensitive runtime configurations for $\mathcal{M}$. We assume given functions $cost_{\mathcal{M}}(t, t')$ and $free_{\mathcal{M}}(t, t')$.

Let $\longrightarrow$ denote the single-step reduction relation of the ABS semantics, defined in Sec. 2.1. A resource-constrained run of an ABS model consists of zero or more applications of a transition relation $\longrightarrow_{\mathcal{M}}$, which is defined by the context

$$\frac{\begin{array}{cc} \textit{mycomp}(o) = id & \textit{cost}_{\mathcal{M}}(o\ \overline{msg}, o'\ \overline{msg}'\ \overline{config}') \leq r \\ o\ \overline{msg} \longrightarrow o'\ \overline{msg}'\ \overline{config}' & r' = r + \textit{free}_{\mathcal{M}}(o\ \overline{msg}, o'\ \overline{msg}'\ \overline{config}') \end{array}}{\{\textit{comp}(id, r)\ o\ \overline{msg}\ \overline{config}\} \longrightarrow_{\mathcal{M}} \{\textit{comp}(id, r')\ o'\ \overline{msg}'\ \overline{config}\ \overline{config}'\}}\ (\textsc{Context})$$

Fig. 4. An operational semantics for resource-constrained deployment components

$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]^{\varepsilon}_{(a \circ l)} \quad vp = l(x) \quad \mathrm{cost}(e) \leq r \quad \mathrm{mycomp}(o) = dc}{\begin{array}{c} dc(r)\ ob(o, C, a, \{l | x := e; s\}, q) \\ \rightarrow dc(r + |vp| - |v|)\ ob(o, C, a, \{l[x \mapsto v] | s\}, q) \end{array}}\ (\textsc{assign1-rsc})$$

Fig. 5. Resource-aware assignment rule, with an object *ob* and deployment component *dc*. The assignment statement is only executed if e can be evaluated with the current r, which is adjusted afterwards.

rule given in Fig. 4. Runtime configurations are extended with the representation of deployment components $comp(id, r)$, where id is the identifier of the component and r its currently available resources. Each object has a field `mycomp`, instantiated to its deployment component at creation time (we omit the redefined object creation rule). Let $\overline{config}$ denote a set of objects and futures. The context rule expresses how an object o may evolve to o' given a set of invocation messages $\overline{msg}$ in the context of a deployment component with r available resources. Since o may consume an invocation message and create new objects, futures, or invocation messages, the right hand side of the rule returns o' with a possibly different set of messages $\overline{msg}'$ and a configuration $\overline{config}'$.

5 Simulation and Experimental Results

To validate the approach presented in this paper, an interpreter for the ABS language was augmented with a resource constraint model that simulates systems with limited memory. The semantics of this ABS interpreter is given in rewriting logic [18] and executes on the Maude platform [10]. Note that the semantics of Section 4, when implemented directly, leads to a significant amount of backtracking in an actual simulation. For this reason, our Maude interpreter was modified to incorporate deployment components and use the costs of Table 1 for the execution of statements. One such modified rule is shown in Figure 5: An assignment to x can only proceed if the cost of evaluating the right-hand side e of the assignment statement is less than the currently free memory r. In this case, x is bound to the new value v, and r is adjusted using Table 1 (here, the difference between v and the previous value vp). All other transition rules which evaluate expressions are modified in the same way.

Simulation results. Deployment component declarations were added to the book shop model described in Example 2, restricting the memory available to all objects of type `Database`, `Agent`, and `Session` (i.e., the server part of the model). Cost functions were computed for all functions in the model, as

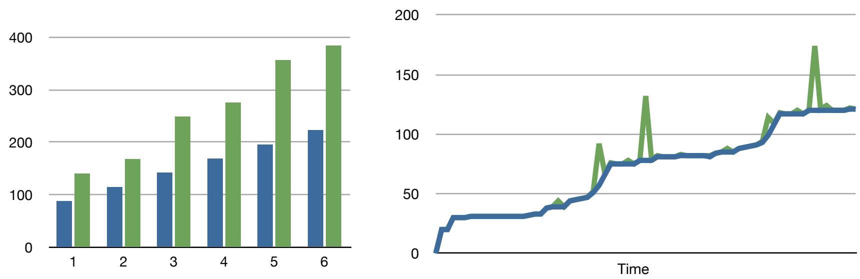

Fig. 6. Final and peak memory use as a function of the size of input (left) and progression of memory use for execution using input size 2 (right).

described in Sec. 3 (UBs for all functions in the book shop model can be found in [5]). With this interpreter, creating a deployment component with too little memory leads to the expected deadlock.

To obtain quantitative results, the interpreter was instrumented to record current memory r and peak memory usage $r + \text{cost}(s)$ during the evaluation of its resource-aware rules. This instrumentation yields both *maximum resource usage* and a *time series* of memory usage for a simulation run. Figure 6 (left) shows the peak intermediate memory usage and memory use at the end of the simulation for various input sizes (i.e., how often to run book orders of constant size). Figure 6 (right) shows the memory use over time of one single run of the model. The "peaks" in the right-hand side graph occur during evaluation of functions with large intermediate memory usage (the blue line represents memory use between execution steps, when the transient memory has been freed again).

6 Related Work

Static cost inference for sequential programming languages has recently received considerable attention. A cost analysis for Java bytecode has been developed in [2], for C++ in [12], and for functional programs in [13]. Our approach for inferring cost for the functional part of ABS is based on [2], which follows the classical approach of [23]. Inference of worst-case UBs on the memory usage of Java like programs with garbage collection is studied in [4]. The analysis accounts for memory freed by garbage collection, and thus infers more tight and realistic bounds. The analysis supports several GC schemes. The analysis of [13] supports inference of memory usage, and accounts for memory freed by destructive matching. In [16] live heap space analysis for a concurrent language has been proposed. However it uses a very limited model of shared memory. Recently, a cost analysis for X10 programs [9] has been developed [3], which infers UBs on the number of tasks that can be running in parallel. The concurrency primitives of X10 are similar to ABS, but X10 is not based on concurrent objects.

Formal resource modeling happens mainly in the embedded domain. For example, Verhoef et al. [22] use the timed VDM++ to model processing time, schedulability and bandwidth resources for distributed embedded systems, but their approach is less general and not used for memory consumption. Johnsen et al. modeled processing resources in the context of deployment components in previous work [15], but this work does not use cost analysis methods. There is not much work combining static cost analysis and simulation to analyze resource usage. However, Künzli et al. [17] combine exact simulation and arrival curves to model processing costs, decreasing the needed simulation time by using arrival curves in their simulations to abstract from some of the components in a SystemC model of specific hardware. In contrast, we use cost analysis to generalize simulations on abstract, formally defined object-oriented models.

7 Discussion

Software is increasingly being developed to be configured for different architectures, which may be restricted in the resources they provide to the software. Therefore, it is interesting to capture aspects of low-level deployment concerns at the abstraction level of a software modeling language. In this paper, we have shown how a formally defined executable concurrent object-oriented modeling language can be extended with a notion of deployment component, which imposes a resource-constraint on the execution of objects in the model.

In order to validate the behavior of the resource-restricted model, we propose to combine static cost analysis with simulations. This combination is achieved by applying static cost analysis to the sequential parts of the modeling language, for which practical cost analysis methods exist, while using simulation for the concurrent part, for which static approaches would lead to a state-space explosion. Thus, the complexity of applying static cost analysis to concurrent executions is avoided, and, in addition, we obtain better results than concrete simulations because the sequential parts of the model are simulated by the worst-case bounds. The technique is demonstrated for memory consumption analysis on an example. The analysis of memory consumption considered here could be strengthened by allowing explicit scheduling and garbage collection policies to be included in the model. This is left for future work.

Another interesting issue is how resource analysis carries over from executable models to generated code. A code generator from ABS to Java is under development that translates user defined abstract data types in ABS into object structures. Hence, the symbolic UBs inferred for memory consumption of the ABS models correspond to bounds on the number of objects in the corresponding Java code. Note that it might not be possible to find similar correlations for other cost models such as the number of execution steps. Another line of interesting future work is to set up actual measurements on generated code and use these results to profile our analysis approach for a given cost model.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning* 42(6):161–203, 2011.

2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. ESOP'07*, LNCS 4421, pages 157–172. Springer, 2007.
3. E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *LCTES*. ACM Press, April 2011.
4. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *ISMM*, ACM Press, 2010.
5. 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. Research Report 403, Dept. of Informatics, Univ. of Oslo, Jan. 2011. `http://einarj.at.ifi.uio.no/Papers/rr403.pdf`
6. F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In *LOPSTR*, LNCS 1207, pages 204–223. Springer, 1997.
7. A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In *TAPSOFT*, LNCS 494.. Springer, 1991.
8. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.
9. P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster computing. In *OOPSLA*, pages 519–538. ACM, 2005.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework*, LNCS 4350. Springer, 2007.
11. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, LNCS 4421, pages 316–330. Springer, 2007.
12. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*, ACM 2009.
13. J Hoffmann, Klaus Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *POPL*, pages 357–370, ACM 2011.
14. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.
15. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In Proc. *ICFEM*, LNCS 6447, pages 646–661. Springer, 2010.
16. M. Kero, P. Pietrzak, and N. J. Live Heap Space Bounds for Real-Time Systems. In *APLAS*, LNCS 6461, pages 287–303. Springer, 2010.
17. S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *DATE*, pages 236–241. European Design and Automation Association, 2006.
18. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
19. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.
20. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
21. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP 2010*, LNCS 6183. Springer, 2010.
22. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. FM 2006*, LNCS 4085, pages 147–162. Springer, 2006.
23. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.
24. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453. ACM Press, 2005