Extending Abstract Behavioral Specifications with Erlang-style Error Handling*

Georg Göri[1], Bernhard K. Aichernig[1],
Einar Broch Johnsen[2], Rudolf Schlatte[2] and Volker Stolz[2]

[1] University of Technology, Graz, Austria
goeri@student.tugraz.at,aichernig@ist.tugraz.at
[2] University of Oslo, Norway
{einarj,rudi,stolz}@ifi.uio.no

1 Introduction

Software modeling languages traditionally abstract from low-level concerns such as the reliability and speed of the deployment architecture, to obtain concise and focused models [6]. However, modern distributed and virtualized systems are increasingly resource-aware, network-aware, and adapt to dynamically changing infrastructure. These concerns need to be captured at the modeling stage, but modeling languages must balance the need for abstraction with the need to express and analyze a system's ability to adapt to its environment.

The abstract behavioral specification (ABS) language targets distributed object-oriented systems [5], but does not currently support fault-tolerant features such as adaptability to distribution failures. This work develops an Erlang execution backend for ABS and extends ABS with error handling capabilities à la Erlang. The resulting extension of ABS combines rollback to invariant states at the object-level with Erlang style process linking and supervision.

2 ABS and Erlang

ABS is a statically typed object-oriented modeling language targeting distributed systems [5]. ABS is based on asynchronous method calls between Concurrent Object Groups (cogs), akin to Actors and concurrent objects. An asynchronous method call creates a new process in the called object which may run in parallel with the continuation of the calling process. The reply from the asynchronous call is placed in a future, a single-assignment global variable that can be shared between objects. Futures support retrieval and checking the availability of a reply. Whereas execution in different cogs happens in parallel, the execution of processes inside a cog is strictly interleaved and controlled by means of cooperative scheduling; i.e., explicit suspension points in the code allow the active process to be suspended and another local process to be activated. Suspension may be conditional; e.g., it can depend on the status of a future. ABS supports a proof theory for concurrent systems by means of local reasoning, based on seeing objects as monitor-like maintainers of class invariants [4]. This proof theory requires that the invariants hold at locally quiescent states; i.e., whenever a process may suspend it must ensure the invariant and whenever a process is scheduled it can assume the invariant.

Erlang is a dynamically typed functional programming language. Concurrency is done by lightweight processes which asynchronously exchange messages but do not share state [1]. Distribution with location transparent message passing is an integral part of Erlang. In addition to a standard exception mechanism, Erlang provides *process linking* to handle distribution and

*Partially funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu).

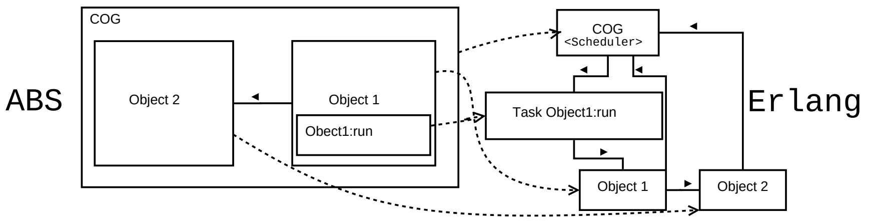

Figure 1: ABS entities mapped to Erlang processes, where solid lines show associations and dashed ones the mapping of components to processes.

runtime errors. A link is a bidirectional relationship between two processes, which guarantees the delivery of an exit signal to one process in case its partner terminates or becomes unreachable. These features allow *supervision* of processes, and enable a style of error handling where processes are allowed to crash and if needed, restart from a previous or initial state.

Our mapping of ABS to Erlang follows the principle that "everything is a process". While adhering to ABS semantics it provides distribution and scalability. Each cog becomes one Erlang process which controls local scheduling such that at most one ABS process can execute at a time. Each ABS process (and the main block) becomes one Erlang process which maintains the local variables. ABS objects become tail-recursive looping processes which handle field access via messages. Figure 1 depicts a runtime view of a model in both languages.

3 Error Handling in Distributed Models

This section presents an error handling schema for ABS. Instead of compensatory actions as in [2], we propose a system that combines error propagation via futures and automatic object-level rollback on failure. This way process linking and recovery operations can be implemented in ABS. Both runtime errors (e.g. division by zero, out of memory) and distribution errors (e.g. connection loss) are represented in the model. We introduce the following language constructs into ABS: a notion of user-defined error types; a generalization of the future mechanism to propagate either return values or errors; a statement **abort** e, which raises an error e and thereby terminates the process; a statement f.**safeget**, which can receive both errors and values from a future f; and a statement **die** which terminates the current object and all its processes.

To enable *error propagation*, ABS futures are enhanced to carry either the normal result of a method invocation or an error. The caller can retrieve the return value with the **get** expression, which will, in case the future carries an error e, lead to an implicit execution of **abort** e. If error handling is desired, the newly introduced **safeget** expression can be used, which will return the result wrapped either as `Value`(< *value* >) or `Error`(e) data constructors. Presented design incorporates the Erlang principles: error propagation with fast failing as default or optional error recovery. The effect of executing **abort** is defined in the following way:

- **Active Object.** If the active object's process evaluates **abort** e, all processes of the object will abort with the error e and the references to this object will become invalid. Further synchronous or asynchronous calls are equivalent to executing **abort** `DeadObject`.
- **Asynchronous Call.** An **abort** e statement terminates the process, stores e in the associated future, and rolls back the object state. This rollback discards all changes since the last scheduling point and thus re-establishes the object invariant.
- **Main Block.** An **abort** here will not be further handled and the execution will terminate.

Towards linking of objects. The addition of a **die** statement, which has an equal effect to an **abort** in an active object (see above), enables us to implement a linking between Objects,

```
class Link(Linkable f,Linkable s){
  Int done=0;
  Unit setup(){
   f!waiton(this,s);
   s!waiton(this,f);
   await done==2;}
  Unit done(){
   done=done+1;}}
```

```
class Linkable() implements Linkable{
  Unit waitOn(Link l,Linkable la){
    Fut<Unit> fut=la!wait();
    l!done();
    await fut?;
    case fut.safeget {
     Error(e) => die e;
}}}
```

Figure 2: Implementation of Links in ABS

which is shown in Fig. 2. Error recovery code can replace the **die** statement in the `Linkable`. Support of the runtime system would drop the need to implement the `Linkable` interface in every class and could call automatically an optional error handling function.

4 Discussion

This paper reports on work connecting ABS to Erlang. The presented Erlang backend seems to scale well to a large number of ABS processes, and can be seen as a step toward a distributed implementation of ABS. This backend forms a basis for adapting Erlang's error handling capabilities to the statically typed object-oriented world of ABS. This paper extends ABS for such error handling; we are currently adapting the rewriting logic semantics of ABS. Next, we will apply the analysis tools of ABS to analyze error handling during the system design.

Two strands of related work are verification systems for Erlang and error handling systems in software models. For example, Castro et al. describe their experience in verifying properties of supervisor trees using McErlang [3]. Although McErlang directly takes Erlang code as input, checking the supervisor needed special patching, due to the lack of time simulation. While in ABS it could stay untouched, because time can be used both in execution and verification. Previous work on errors in ABS models proposes a compensation mechanism inspired by web services; it would be interesting to see how this approach could be integrated with the work reported here. The proposed model-based error handling allows the generation of concise test models for fault-tolerant distributed systems which can be explored by test-driven simulations. An interesting extension to the rollback-mechanism is to relate it to transactions. Another line of future work is to adapt Erlang's "failure free" way of message passing, where communication errors are ignored in send/receive events, and instead handled by monitors: in our setting, sending to an invalid process will lead to an abort immediately.

References

[1] Armstrong, J. *Programming Erlang.* Pragmatic Bookshelf, 2007.

[2] Johnsen, E. B., Lanese, I., and Zavattaro, G. Fault in the future. In *Coordination Models and Languages*, *LNCS* 6721, pages 1–15. Springer, 2011.

[3] Castro, D., Gulias V. M., Benac Earle, C., Fredlund, L.-Å., and Rivas, S. A case study on verifying a supervisor component using McErlang. In *PROLE 2010*, *ENTCS* 271, pages 23–40, 2011.

[4] Chang Din, C., Dovland, J., Johnsen, E. B., and Owe, O. Observable behavior of distributed systems: Component reasoning for concurrent objects. *J. of Log. and Alg. Prog.*, 81:227–256, 2012.

[5] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. ABS: A core language for abstract behavioral specification. In *FMCO*, *LNCS* 6957, pages 142–164. Springer, 2012.

[6] Kramer, J. Is abstraction the key to computing? *Comm. ACM*, 50(4):36–42, 2007.