SoOSiM: Operating System and Programming Language Exploration

Christiaan Baaij, Jan Kuper
Computer Architecture for Embedded Systems,
Department of EEMCS, University of Twente,
Postbus 217, 7500AE Enschede, The Netherlands
Email: {c.p.r.baaij;j.kuper}@utwente.nl

Lutz Schubert
HLRS – University of Stuttgart,
Department of Intelligent Service Infrastructures,
Nobelstr. 19, D-70569 Stuttgart, Germany
Email: schubert@hlrs.de

***Abstract*—SoOSiM is a simulator developed for the purpose of exploring operating system concepts and operating system modules. The simulator provides a highly abstracted view of a computing system, consisting of computing nodes, and components that are concurrently executed on these nodes. OS modules are subsequently modelled as components that progress as a result of reacting to two types of events: messages from other components, or a system-wide tick event. Using this abstract view, a developer can quickly formalize assertions regarding the interaction between operating system modules and applications.**

We developed a methodology on top of SoOSiM that enables the precise control of the interaction between a simulated application and the operating system. Embedded languages are used to model the application once, and different interpretations of the embedded language constructs are used to observe specific aspects on application's execution. The combination of SoOSiM and embedded languages facilitates the exploration of programming language concepts and their interaction with the operating system.

I. Introduction

Simulation is a commonly used tool in the exploration of many design aspects of a system: ranging from feasibility aspects to gathering performance information. However, when tasked with the creation of new operating system concepts, and their interaction with the programmability of large-scale systems, existing simulation packages do not seem to have the right abstractions for fast design exploration [1], [2] (ref. Section IV). The work we present in this paper has been created in the context of the S(o)OS project [3]. The S(o)OS project aims to research OS concepts and specific OS modules, which aid in scalability of the complete software stack (both OS and application) on future many-core systems. One of the key concepts of S(o)OS is that only those OS modules needed by a application thread, are actually loaded into the (local) memory of a Core / CPU on which the thread will run. This execution environment differs from contemporary operating systems where every core runs a complete copy of the (monolithic) operating system.

A basic requirement that the S(o)OS project has towards any simulator, are the facilities to straightforwardly simulate the instantiation of application threads and OS modules. Aside from the fact that the S(o)OS-envisioned system will be dynamic as a result of loading OS modules on-the-fly; large-scale systems also tend to be dynamic in the sense that computing nodes can (permanently) disappear (failure), or appear (hot-swap). Hence, the simulator has to facilitate the straightforward creation and destruction of computing elements. The current need for a simulator rests mostly in formalizing the S(o)OS concept, and examining the interaction between the envisioned OS modules and the application threads. As such, being able to extract highly accurate performance figures from a simulated system is not a key requirement. It should however facilitate the ability to observe all interactions between application threads and OS modules should. Additionally, a user should able to *zoom in* on particular aspects of the behaviour of an application: such as memory access, messaging, etc.

This paper describes a new simulator, *SoOSiM* , that meets the above requirements. We elaborate on the main concepts of the simulator in Section II, and show how OS modules interact with each other, and with the simulator. Section III describes the use of embedded languages for the creation of applications running in the simulated environment. The simulation engine, the graphical user interface, and embedded language environment are all written in the functional programming language Haskell [4]; this means that all code listings in this paper also show Haskell code. Due to limitation in the number of pages, we are not be able to elaborate every Haskell notation; the code examples are intended to support the validity of the presented concepts. Section IV compares SoOSiM to existing simulation frameworks, and lists other related work. Section V enumerates our experiences with SoOSiM, and Section VI discusses potential future work.

II. Abstract System Simulator

The purpose of SoOSiM is mainly to provide a platform that allows a developer to observe the interactions between OS modules and application threads. It is for this reason that the simulated hardware is highly abstract. In SoOSiM, the hardware platform is described as a set of nodes. Each *node* represents a physical computing object: such as a core, complete CPU, memory controller, etc. Every node has a local memory of potentially infinite size. The layout and connectivity properties of the nodes are not part of the system description.

This work is supported through the S(o)OS project, sponsored by the European Commission under FP7-ICT-2009.8.1, Grant Agreement No. 248465.
SoOSiM is available on: http://hackage.haskell.org/package/SoOSiM

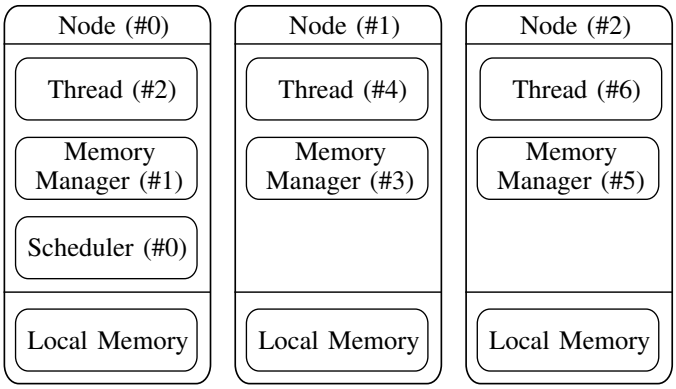

Fig. 1. Abstracted System

Each *node* hosts a set of components. A *component* represents an executable object: such as a thread, application, OS module, etc. Components communicate with each other either using direct messaging, or through the local memory of a node. Having both explicit messaging and shared memories, SoOSiM supports the two well known methods of communication. Components have a (hidden) message queue, because:

- Multiple components can send messages to the same component concurrently.
- A component can receive messages while it is waiting for a response from another component.

All components in a simulated system, even those hosted within the same node, are executed concurrently (from the component's point of view). The simulator poses no restrictions as to which components can communicate with each other, nor to which node's local memory they can read from and write to. A schematic overview of an example system can be seen in Figure 1.

The simulator progresses all components concurrently in one discrete step called a $tick$. During a *tick*, the simulator passes the content that is at the head of the message queue of each individual component. If the message queue of a component is empty, a component will be executed with a *null* message. If desired, a component can inform the simulator that it does not want to receive these *null* messages. In that case the component will not be executed by the simulator during a *tick*.

A. OS Component Descriptions

Components of the simulated system are, like the simulator core, also described in the functional programming language Haskell. This means that each component is described as a function. In case of SoOSiM, such a function is not a simple algebraic function, but a function executed within the context of the simulator. The Haskell parlance for such a computational context is a *Monad* [4, Chapter 14], the term we will use henceforth. Because the function is executed within the monad, it can have *side-effects* such as sending messages to other components, or reading the memory of a local memory. In addition, the function can be temporarily suspended at (almost) any point in the code. SoOSiM needs to be able to suspend the execution of a function so that it may emulate synchronous messaging between components, a subject we will further elaborate later on.

We describe a component as a function that, as its first argument, receives a user-defined internal state, and as its second argument a value of type $Event\ a$. The result of this function will be the (potentially updated) internal state. Values of type $Event\ a$ can either be:

- A message from another component, where 'a' represents the datatype of the content of the message.
- A *null* message.

We thus have the following type signature for a component:

$$component :: State \rightarrow Event\ a \rightarrow Sim\ State$$

The Sim annotation on the result type means that this function is executed within the simulator monad. The user-defined internal state can be used to store any information that needs to perpetuate across simulator *ticks*.

To include a component description in the simulator, the developer will have to create a so-called *instance* of the $ComponentInterface$ *type-class*. A *type-class* [4, Chapter 6] in Haskell can be compared to an interface definition as those known in object-oriented languages. An *instance* of a *type-class* is a concrete instantiation of such an interface. SoOSiM users should use a singleton datatype to uniquely label the interface description of a component. The $ComponentInterface$ consists of the following values to completely define a component:

- The datatype representing the internal state.
- The datatype of the received messages.
- The datatype of the send messages.
- The initial internal state of the component.
- The unique name of the component.
- The monadic function describing the behaviour.

We stress again that we are aiming at a high level of abstraction for the behavioural descriptions of our OS modules, where the focus is mainly on the interaction with other OS modules and application threads.

B. Interaction with the simulator

Components have several functions at their disposal to interact with the simulator and consequently interact with other components. The available functions are the following:

- $createComponent$ instantiates a new component on a specified node.
- $invoke$ sends a message to another component, and waits for the answer. Whenever a component uses this function it will be suspended by the simulator. Several simulator ticks might pass before the callee sends a response. Once the response is available the simulator resumes the execution of the calling component.
- $invokeAsync$ sends a message to another component, and registers a handler with the simulator to process the response. In contrast to $invoke$, using this function will *not* suspend the execution of the component.
- $respond$ sends a message to another component as a response to an invocation.

- *yield* informs the simulator that the component does not want to receive *null* messages.
- *readMem* performs a read at a specified address of a node's local memory.
- *writeMem* writes a value at a specified address of a node's local memory.
- The *componentLookup* function performs a lookup of the unique identifier of a component given a specified interface.

Components have a *ComponentId* that is a unique number corresponding to a specific instance of a component. The knowledge of the unique *ComponentId* of the specific instance is needed to invoke a component. To give a concrete example, using the system of Figure 1 as our context: *Thread(#6)* wants to invoke the instance of the *MemoryManager* that is running on the same Node (#2). As *Thread(#6)* was not involved with the instantiation of that OS module, it has no idea what the specific *ComponentId* of the memory manager on Node #2 is. It does however know the interface-label of the memory managers, so it can use the *componentLookup* function to find the *MemoryManager* with ID #5 that is running on Node #2.

C. Example OS Component: Memory Manager

This subsection demonstrates the use of the simulator API, taking the *Read* code-path of the memory manager module as an example. The memory manager takes care that the reads or writes of a global address end up in the correct node's local memory. As part of its internal state the memory manager keeps a lookup table. This lookup table states whether an address range belongs to the local memory of the node that hosts the memory manager, or whether that address is handled by a memory manager on another node. An entry of the lookup table has the following datatype:

```
data Entry = EntryC
  { base  :: Int
  , scope :: Int
  , scrId :: Maybe ComponentId
  }
```

The fields *base* and *scope* together describe the memory address range defined by this entry. The *srcId* tells us whether the range is hosted on the node's local memory, or whether another memory manager is responsible for the address range. If the value of *scrId* is *Nothing* the address is hosted on the node's local memory; if *srcId* has the value *Just cmpId*, the memory manager with ID *cmpId* is responsible for the address range.

Listing 1 highlights the Haskell code for the read-logic of the memory manager. Lines 1, 2, and 3 show the type signature of the function defining the behaviour of the memory manager. On line 4 we use pattern-matching, to match on a *Message* event, binding the values of the message content, and the identification of the caller, to *content* and *caller* respectively. We examine the *content* on line 4, and only continue when it is a *Read* message (indicated by the vertical bar |). If it is a *Read* message, we bind the value of the address to the name *addr*. On line 7 we lookup the address range entry which encompasses *addr*. Line 8 starts a **case**-statement discriminating on the value of the *srcId* of the entry. If the *srcId* is *Nothing* (line 9-12), we read the node's local memory using the *readMem* function, *respond* to the caller with the read value, and finally *yield* to the simulator. When the address range is handled by a *remote* memory manager (line 13-17), we *invoke* that specific memory manager module with the read request and wait for a response. We remark that many simulator cycles might pass between the invocation and the return, as the *remote* memory manager might be processing many requests. Once we receive the value from the *remote* memory manager, we *respond* to the original caller forwarding the received value.

Note that the functions *invoke* and *respond* each receive, as their first argument, the singleton-datatype that was used to label the memory manager interface. This label is used to access the *Receive* and *Send* datatype fields of the interface, and statically ensures that we only send and receive datatypes that correspond to the interface of the memory manager.

Listing 1 Read logic of the Memory Manager

```
memoryManager :: MemState                                  1
              → Event MemCommand                           2
              → Sim MemState                               3
memoryManager s (Message content caller)                   4
  | (Read addr) ← content                                  5
  = do                                                     6
    let entry = addressLookup s addr                       7
    case (srcId entry) of                                  8
      Nothing → do                                         9
        addrVal ← readMem addr                             10
        respond MemoryManager caller addrVal               11
        yield s                                            12
      Just remote → do                                     13
        response ← invoke MemoryManager                    14
                   remote content                          15
        respond MemoryManager caller response              16
        yield s                                            17
  | (Write addr val) ← content                             18
  = do                                                     19
    ...                                                    20
```

D. Simulator GUI

The state of a simulated system can be observed using the SoOSiM GUI, of which a screenshot is shown in Figure 2. The GUI allows you to run and step through a simulation at different speeds. On the screenshot we see, at the top, the toolbar controlling the simulation, in the middle, a schematic overview of the simulated system, and specific information belonging to a selected component at the bottom. Different colours are used to indicate whether a component is active, waiting for a response, or idle. The *Component Info* box shows both static and statistical information regarding a selected component. Several statistics are collected by the simulator, including the number of simulation cycles spent in a certain state (active / idle / waiting), messages sent and received, etc.

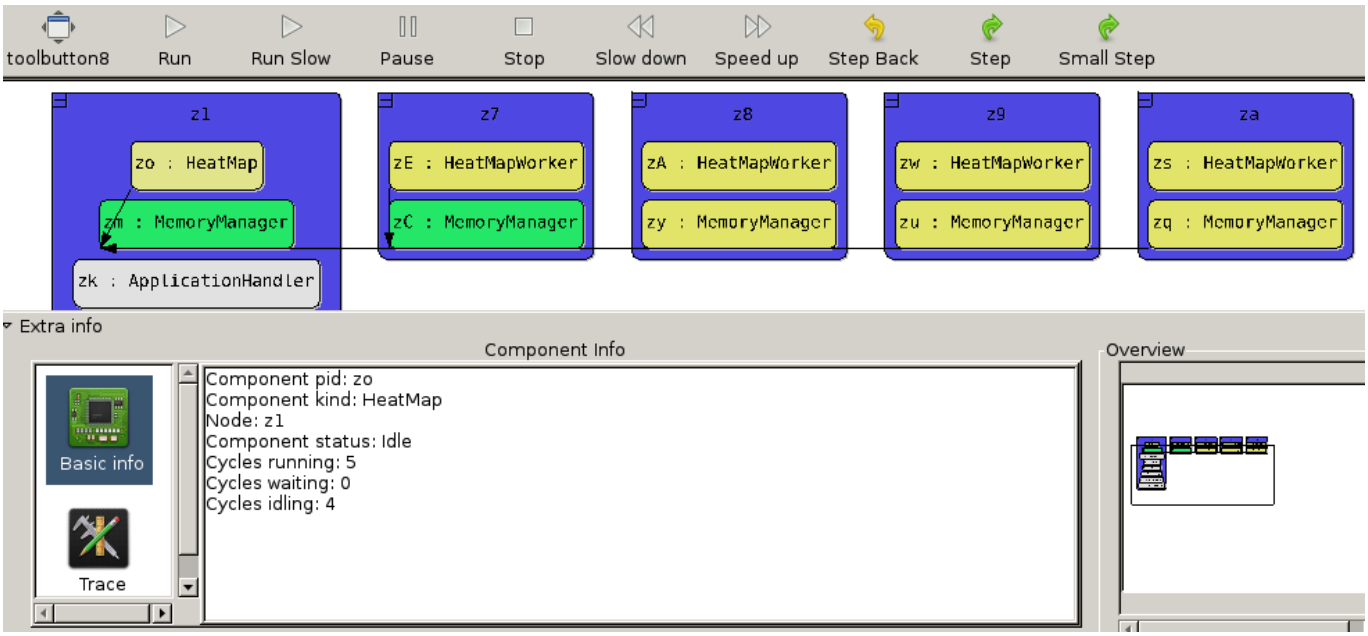

Fig. 2. Simulator GUI

These statistics can be used to roughly evaluate the performance bottlenecks in a system. For example, when OS module 'A' has mostly active cycles, and components 'B'-'Z' are mostly waiting, one can check if components 'B'-'Z' were indeed communicating with 'A'. If this happens to be the case, then 'A' is indeed a bottleneck in the system. A general rule-of-thumb for a well performing system is when OS modules have many *idle* cycles, and application threads have mostly *active* cycles.

III. Embedded Programming Environment

One of the reasons to develop SoOSiM is to observe the interaction between applications and the operating system. Additionally, we want to explore programming language concepts intended for parallel and concurrent programming, and how they impact the entire software stack. For this purpose we have developed a methodology on top of SoOSiM, that uses embedded languages to specify the applications. Our methodology consists of two important aspects:

- The use of embedded (programming) languages to define an application.
- Defining different interpretations for such an application description, allowing a developer to observe different aspects of the execution of an application.

A. Embedded Languages

An *embedded language* is a language that can be used from within another language or application. The language that is embedded is called the *object* language, and the language in which the *object* language is embedded is called the *host* language. Because the *object* language is *embedded*, the *host* language has complete control over any terms / expressions defined within this *object* language. There are multiple ways of representing embedded languages, for example as a string, which must subsequently be parsed within the *host* language.

Haskell has been used to host many kinds of embedded (domain-specific) languages [5]. The standard approach in Haskell is not to represent *object* terms as strings, but instead use data-types and functions. To make this idea more concrete, we present the recursive Fibonacci function, defined using one of our self-defined *embedded* functional languages, in Listing 2.

Listing 2 Call-by-Value Fibbonaci

```
fib :: Symantics repr ⇒ repr (IntT ↠ IntT)                          1
fib = fix $ λf →                                                    2
  fun $ λn →                                                        3
    nv 0 $ λn1 →                                                    4
    nv 0 $ λn2 →                                                    5
    nv 0 $ λn3 →                                                    6
      n1 =: n `seq`                                                 7
      if_ (lt (drf n1) 2)                                           8
        1                                                           9
        (n2 =: (app f (drf n1 − 1)) `seq`                           10
         n3 =: (app f (drf n1 − 2)) `seq`                           11
         drf n2 + drf n3                                            12
        )                                                           13
```

All functions printed in **bold** are language constructs in our *embedded language*. Additionally the =: operator is also one of our *embedded* language constructs; the numeric operators and literals are also overloaded to represent embedded terms. To give some insight as to how Listing 2 represents the recursive Fibonacci function, we quickly elaborate each of the lines.

The type annotation on line 1 tells us that we have a function defined at the *object*-level (↠) with an *object*-level integer ($IntT$) as argument and an *object*-level integer (IntT) as result.

Line 2 creates a fixed-point over f, making the recursion of our embedded Fibonacci function explicit. On line 3 we define a function parameter n using the **fun** construct. We remark that we use Haskell binders to represent binders in our *embedded language*. On line 4-6 we introduce three mutable references, all having the initial integer value of 0. We assign the value of n to the mutable reference $n1$ on line 7. On line 8 we check if the dereferenced value of $n1$ is less than 2; if so we return 1 (line 9); otherwise we assign the value of the recursive call of f with $(n1 - 1)$ to $n2$, and assign the value of the recursive call of f with $(n1 - 2)$ to $n3$. We subsequently return the addition of the dereferenced variables $n2$ and $n3$.

We must confess that there is some syntactic overhead as a result of using Haskell functions and datatypes to specify the language constructs of our *embedded* language; as opposed to using a string representation. However, we have consequently saved ourselves from many implementation burdens associated with embedded languages:

- We do not have to create a parser for our language.
- We can use Haskell bindings to represent bindings in our own language, avoiding the need to deal with such *tricky* concepts as: symbol tables, free variable calculation, and capture-free substitution.
- We can use Haskell's type system to represent types in our embedded language: meaning we can use Haskell's type-checker to check expressions defined in our own embedded language.

B. Interpreting an Embedded Language

We mentioned the concept of *type-classes* when we discussed the process of including a component description in the simulator. Following the *final tagless* [6] encoding of embedded languages in Haskell, we use a type-class to define the language constructs of our mini functional language with mutable references. A partial specification of the $Symantics$ (a pun on *syntax* and *semantics* [6]) type-class, defining our *embedded language*, is shown in Listing 3.

Listing 3 Embedded Language - Partial Definition. Viz. [6]

```
class Symantics repr where                                          1
  fun :: (repr a → repr b) → repr (a ↠ b)                           2
  app :: repr (a ↠ b) → repr a → repr b                             3
  drf :: repr (Ref a) → repr a                                      4
  (=:) :: repr (Ref a) → repr a → repr Void                         5
```

We read the types of our language definition constructs as follows:

- **fun** takes a *host*-level function from $object$-type a to $object$-type b ($repr\ a \rightarrow repr\ b$), and returns an *object*-level function from a to b ($a \twoheadrightarrow b$).
- **app** takes an *object*-level function from a to b, and applies this function to an *object*-term of type a, returning an *object*-term of type b.
- **drf** dereferences an $object$-term of type "reference of" a (written in Haskell as $Ref\ a$), returning an *object*-term of type a.
- $(=:)$ is operator that updates an *object*-term of type "reference of" a, with a new *object*-value of type a, returning an *object*-term of type $Void$.

To give a desired interpretation of an application described by our embedded language we simply have to implement an instance of the $Symantics$ type-class. These interpretations include pretty-printing the description, determining the size of expression, evaluating the description as if it were a normal Haskell function, etc.

In the context of this paper we are however interested in *observing* (specific parts of) the execution of an application inside the SoOSiM simulator. As a running example, we show part of an instance definition that observes the invocations of the memory manager module upon dereferencing and updating mutable references:

Listing 4 Observing Memory Access - Partial definition

```
instance Symantics Sim where                                        1
  drf x = do                                                        2
    i     ← foo x                                                   3
    mmId ← componentLookup MemoryManager                            4
    invoke MemoryManager mmId (Read i)                              5
  x =: y = do                                                       6
    i     ← foo x                                                   7
    v     ← bar y                                                   8
    mmId ← componentLookup MemoryManager                            9
    invoke MemoryManager mmId (Write i v)                           10
```

We explained earlier that the simulator *monad* (Sim) should be seen as a computational context in which a function is executed. By making our simulator monad the computational **instance** (or environment) of our embedded language definition, we can now run the applications defined with our embedded language inside the SoOSiM simulator. Most language constructs of our embedded language will be implemented in such a way that they behave like their Haskell counterpart. The constructs where we made minor adjustments are the **drf** and (=:) constructs, which now enact communication with our *Memory Manager* OS module. By using the $invoke$ function, our application descriptions are also suspended whenever they dereference or update memory locations, as they have to wait for a response from the memory manager. Using the SoOSiM GUI, we can now observe the communication patterns between the applications described in our embedded language, and our newly created OS module.

C. Further Extensions and Interpretations

The use cases of embedded languages in the context of our simulation framework extend far beyond the example given in the previous subsection. We can for example easily extend our language definition with constructs for parallel composition, and introduce blocking mutable references for communication

between threads. An initial interpretation (in the form of a type-class instance) could then be sequential execution, allowing for the simple search of algorithmic bugs in the application. A second instance could then use the Haskell counterparts for parallel composition and block mutable variables to mimic an actual concurrent execution. A third instance could then interact with OS modules inside a SoOSiM simulated system, allowing a developer to observe the interaction between our new language constructs and the operating system.

We said earlier that one of the interpretations of an embedded language description could be a pretty-printed string-representation. Following up on the idea of converting a description to a datatype, we can also interpret our application description as an abstract syntax tree or even a dependency graph. Such a dependency graph could then be used in another instances of our embedded language that facilitates the automatic parallel execution of independent sub-expressions. Again, we can hook up such an instance to our simulator monad, and observe the effects of the distribution of computation and data, as facilitated by our simulated operating system.

IV. Related Work

COTSon [1] is a full system simulator, using an emulator (such as SimNow) for the processor architecture. It allows a developer to execute normal x86-code in a simulated environment. COTSon is far too detailed for our needs, and does not facilitate the easy exploration of a complete operating system.

OMNeT++ [2] is a C++-based discrete event simulator for modelling distributed or parallel system. Compared to SoOSiM, OMNeT++ does not allow the straightforward creation of new modules, meaning the distribution of modules is static. OMNeT++ is thus not meeting our simulation needs to dynamically instantiate new OS modules and application threads.

House [7] is an operating system built in Haskell; it uses a Haskell run-time system allowing direct execution on bare metal. OS modules are executed with the *Hardware* monad, comparable to our *Simulator* monad, allowing direct interaction with real hardware. Consequently, OS modules in House must be implemented in full detail, meaning this approach is not suitable for our exploration needs.

Barrelfish [8] is an OS in which embedded languages are used, amongst other purposes, to define driver interfaces. These embedded languages are also implemented in Haskell. The approach used in Barrelfish is however to create parsers for their embedded languages so that they may have a *nicer* syntax, inducing an additional implementation burden.

V. Conclusions

Although the SoOSiM simulator is still considered work in progress, it has already allowed us to formalize the interactions between the different OS modules devised within the S(o)OS [3] project. We believe that this is the strength of our simulator's approach: the quick exploration and formalization of system concepts. Fast exploration is achieved by the highly abstracted view of SoOSiM on the hardware / system. However, having to actually program all our OS modules forces us to formalize the interactions within the system; exposing any potential flaw not discovered by an informal (text-based) description of the operating system.

By using embedded languages to program applications that run in our simulated environment, we attain complete control of its execution. By using specific interpretations of our embedded language, we can easily observe specific parts (such as memory access) of an application's execution. Using Haskell functions to specify our embedded language constructs saves us from a high implementation burden usually associated with the creation of the tools / compilers for programming languages.

VI. Future Work

At the moment, the simulation core of SoOSiM is single-threaded. We expect that as we move to the simulation of systems with 10's to 100's of computing nodes, that the single threaded approach can become a performance bottleneck. Although individual components are susceptible for parallel execution, the communication between components is problematically non-deterministic. We plan to use Haskell's implementation of software transactional memory (STM) to safely deal with the non-deterministic communication and still achieve parallel execution.

We will additionally explore the use of embedded languages, in the domain of operating system and programming language design, further. Within the context of the S(o)OS project, we intend to add both explicit parallel composition to our embedded language definition, and implicit parallel interpretation of data-independent sub-expressions. We also intend to implement software transactional memory constructs, and investigate their interaction with the operating system.

Acknowledgements

The authors would like to thank Ivan Perez for the design and implementation of the SoOSiM GUI.

References

[1] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: Infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, Jan. 2009.

[2] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proceedings of Simutools '08*, ICST, Brussels, Belgium, 2008, pp. 1–10.

[3] L. Schubert, A. Kipp, B. Koller, and S. Wesner, "Service-oriented operating systems: future workspaces," *Wireless Communications, IEEE*, vol. 16, no. 3, pp. 42–50, june 2009.

[4] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[5] Haskell Wiki. (2012, May) Embedded domain specific language. [Online]. Available: http://www.haskell.org/haskellwiki/Embedded_domain_specific_language

[6] J. Carette, O. Kiselyov, and C.-c. Shan, "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages," *J. Funct. Program.*, vol. 19, no. 5, pp. 509–543, Sep. 2009.

[7] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach, "A Principled Approach to Operating System Construction in Haskell," in *Proceedings of ICFP '05*. New York, NY, USA: ACM, 2005, pp. 116–128.

[8] P.-E. Dagand, A. Baumann, and T. Roscoe, "Filet-o-Fish: practical and dependable domain-specific languages for OS development," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 35–39, Jan. 2010.