Using Abstract State Machines at Microsoft: A Case Study

Mike Barnett, Egon Börger*, Yuri Gurevich,
Wolfram Schulte, and Margus Veanes

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{mbarnett, boerger, gurevich, schulte, margus}@microsoft.com

Abstract. Our goal is to provide a rigorous method, clear notation and convenient tool support for high-level system design and analysis. For this purpose we use abstract state machines (ASMs). Here we describe a particular case study: modeling a debugger of a stack based runtime environment. The study provides evidence for ASMs being a suitable tool for building *executable* models of software systems on various abstraction levels, with precise refinement relationships connecting the models. High level ASM models of proposed or existing programs can be used throughout the software development cycle. In particular, ASMs can be used to model inter component behavior on any desired level of detail. This allows one to specify application programming interfaces more precisely than it is done currently.

1 Introduction

This paper describes a case study on the use of ASMs as support for design and analysis of software at Microsoft. In order to use ASMs we needed a tool for executing ASMs integrated with the Microsoft programming environment, in particular with the *Component Object Model* or *COM* [5]. We developed a prototype called AsmHugs [8] by extending the Hugs system [2] which is an implementation of the lazy functional programming language Haskell. AsmHugs is in many ways similar to AsmGofer [1] but Hugs enabled us to use H/Direct [6, 7] for integration with COM. A detailed technical report of this case study is in preparation [3].

1.1 What is the case study about?

We present a model for a command-line debugger of a stack-based runtime environment. The model was reverse-engineered from the debugger which is written in C++ and which uses a particular application programming interface (API). We have in fact three models of the debugger, each at a different level of abstraction. They form a refinement hierarchy where each refinement relationship is a procedural abstraction.

* Visiting researcher from Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy, boerger@di.unipi.it

1.2 Why this particular case study?

Our motivation was to study the applicability (and integrability) of ASMs to Microsoft software. Software at Microsoft is generally written in modules, or components. These often form client/server relationships where a component may function both as a server and as a client. A client and a server interact solely through an established API. An API is a set of procedures to which all communication is restricted. The debugger in question is an application program for end-users. It is a client of the *debug services*.

Suppose you want to understand how the API works without reading the code. You have a specification in the Interface Definition Language (IDL [5]) which gives you only the relevant *signatures*. The inner workings of the API's methods are hidden; the only additional information you may find is an informal natural-language description. Such descriptions may be incomplete and become often inconsistent with the code, as the code evolves over time. Obviously, there is no way to formally enforce correspondence between code and its natural-language description. The two main problems with using program code itself as documentation are these. First, the code is usually huge and includes too many irrelevant details. Second, the code might not be available for proprietary reasons.

In this study, we model a particular program but our interests are broader: how to use ASMs to better specify the APIs by which different components interact. An ASM model provides the missing information at the appropriate level of abstraction so that the user of a given component can understand both the behavior of the component and the protocol that the user is supposed to follow in order to exploit the behavior. Each method is specified as a rule, and the valid patterns of calls are reflected in the state.

As part of a broader project, we also built an ASM model of the other side of the API: the debug services (to be described in a forthcoming report); the debugger model was a valuable source in that context.

1.3 COM

Microsoft software is usually composed of COM components. These are really just static containers of methods. In your PC, you will find dynamic-link libraries (DLLs); a library contains one or more components (in compiled form). COM is a language-independent as well as machine-independent binary standard for component communication. An API for a COM component is composed of *interfaces*; an interface is an access point through which one accesses a set of methods. A client of a COM component never accesses directly the component's inner state, or even cares about its identity; it only makes use of the functionality provided by different methods behind the interface (or by requesting a different interface).

1.4 Integration with COM

Writing an executable model for a client of a COM component requires that either you model also the server side of the interfaces (and then maybe the COM components for which that server is a client, and so on) or you integrate your model into the COM environment and thus make it possible for your model to actually call the COM component. We do the latter. H/Direct provides a means for a Haskell program to communicate with COM components both as a client and as a server. For the debugger model, both modes (the client and the server) are needed. The server mode is needed because the debug services API specifies the *callback* interface that the client must implement. The debug services use the callback interface to make asynchronous procedure calls to the model. However, Hugs is a sequential system. In order to use it in an asynchronous multi-threaded COM environment, a modification of the Hugs runtime was required. Other modifications of the Hugs runtime were needed to allow the outside world to invoke Hugs functions.

All three of the debugger models in the case study, even the most refined one (the *ground* model), are sequential ASMs (enriched with some Haskell functionalities). The ground model communicates with the outside world. The question arises how to view that communication in ASM terms. Calls from the ground model to the debug services are invocations of external functions. Calls in the other direction (the callbacks) happen to be more specific: callbacks are updates of nullary monitored functions.

2 The Case Study

The case study is to model a sample debugger of a particular runtime environment. The main goal of this debugger is to illustrate a correct way to use the debugging services provided by that runtime. See Figure 1. Nonetheless, the debugger has more than 30k lines of C++ code and exhibits complex behavior, partly due to the involvement of several asynchronous agents (the user, the runtime and the operating system), but mostly because of the complexity of the debug services that expose about 50 interfaces and 250 methods.

A careful analysis of the debugger, mainly by analyzing the source code, led us to a refinement hierarchy consisting of three models. By analyzing the causal dependencies of different actions we realized that the complications due to the seemingly asynchronous behaviour could be completely avoided, which enabled us to model the debugger by a *sequential* ASM. The design decisions which are reflected by the resulting debugger model were later validated by running the model as a replacement of the actual debugger.

1. *Control model.* The abstraction level of this model is at the control level. The user can enter a command if the debugger is in a mode where the user has a prompt. The runtime environment can issue a callback to the debugger only if the latter is expecting a callback. At this level, the only effect of a callback or user command is to change the control state.

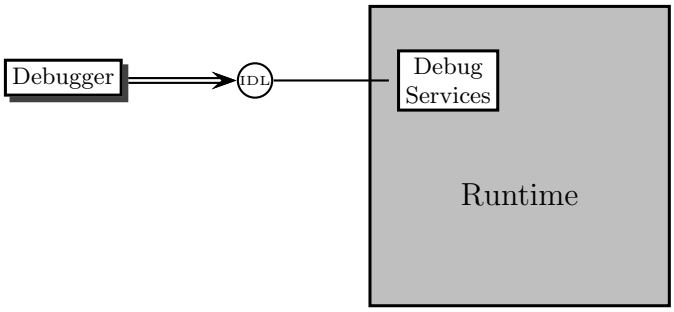

Fig. 1. Debugger and the runtime debug services.

2. *Object model.* This model reflects the *compile time* or *static* structure of the underlying architecture, namely that modules contain classes, classes contain functions, functions contain code, etc. Furthermore, it provides a restricted view of the *run time* structure, namely that processes contain threads, threads contain frames, etc. At this level exactly those commands are modeled that are intimately connected to the compile time structure and to the restricted view of the run time structure, such as execution control, breakpoint control, and stepping commands.
3. *Ground Model.* This model has the same core functionality as the debugger. It provides a more detailed view of the run time structure than the object model. User commands dealing with inspection of the run time stack, and contents of individual frames, like inspection of specific object fields, are modeled here.

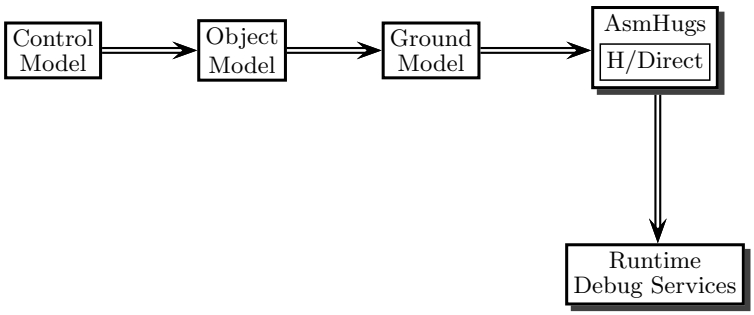

Fig. 2. Connections between the models and the runtime.

In principal, all models are executable, either without the runtime or through the refinements using the actual runtime (see Figure 2). However, in the former case, a proper simulation harness has to be provided. An example of a test harness by using a "wizard" is illustrated in Section 3.2. In a related project, we constructed an ASM model that approximates the runtime. This model could be used here to embody the runtime in the harness.

When executing the control model using the runtime, each action is interpreted by AsmHugs via the indicated refinements; COM connectivity is provided by H/Direct.

3 Control Model of the Debugger

In this model we consider the main control modes of the debugger. These modes reflect who has the control: the user, the runtime, or the debugger itself. When the user has control, the debugger presents a prompt and waits for input. The debugger waits for a response from the runtime when the runtime has the control. Once the runtime has responded with an event, the debugger has control and has to decide what mode to transit to. As we will see, this decision is based on particular properties of the event and further input from the runtime.

Although the communication between the real debugger and the runtime is asynchronous, the documentation of the debug API specifies that the runtime issues at most one asynchronous event at a time; before issuing the next event the runtime must receive an acknowledgement from the debugger. Furthermore, the communication protocol between the debugger and the runtime ensures that at most one of them may have control. Therefore, *sequential* ASMs suffice for our modeling purposes.

The remainder of the section is structured as follows. First, we will lay out the control model ASM in full detail, introducing the necessary state components and the rules as needed. Second, we will run a particular user scenario that will indicate a possible discrepancy between the runtime and the model. Finally, we will remedy the model.

This example shows a typical use of a high level model during the design phase in a software development cycle.

3.1 Control Model ASM

The debugger can be in one of four modes. In `Init` mode the debugger either hasn't been started yet, or it has been terminated. In `Break` mode the user has the control. In `Run` mode the runtime has the control. In `Break?` mode the debugger has the control. The dynamic ASM function, `dbgMode`, records the mode of the debugger in the current state; it has the initial value `Init`. The top level rule of the debugger is `dbg`. Below, `do` is short for `do in-parallel`.

```
dbgMode = initVal Init
dbg = do
```

```
  if dbgMode == Break or dbgMode == Init then handleCommands
  if dbgMode == Run                      then handleResponses
  if dbgMode == Break?                   then handlePendingEvents
```

There are two monitored functions, `command` and `response`, that are updated by the user and the runtime, respectively. Initially, both monitored functions have a value that indicates that no input has been entered by either.

```
command = initVal "nothing"
response = initVal "nothing"
```

User commands We can partition user commands into three groups: commands for starting and quitting the debugger, commands that hand the control over to the runtime (e.g. execution control and stepping commands), and commands that do not affect the control mode (e.g. state inspection and breakpoint setting).

The user can issue commands only if the debugger is either in `Init` mode or in `Break` mode. In the first case, the only meaningful action is to start the debugger.

```
handleCommands = do onStart
                    onExit
                    onBreakingCommand
                    onRunningCommand
                    command := "nothing"

onStart = if dbgMode == Init  and command == "start"
            then do doCommand("start")
                    dbgMode := Break
```

In `Break` mode the debugger handles normal debugging commands and it may switch to `Run` mode or back to `Init` mode, depending on the command.

```
onExit  = if dbgMode == Break and command == "exit"
             then do doCommand("exit")
                     dbgMode:= Init

onBreakingCommand = if dbgMode == Break and isBreakingCommand(command)
                      then do doCommand(command)

onRunningCommand  = if dbgMode == Break and isRunningCommand(command)
                      then do doCommand(command)
                              dbgMode:= Run
```

Firing of execution control commands and stepping commands implies that the control is handed over to the runtime.

```
isRunningCommand(x) = x in? {"run <pgm>", "continue", "kill",
                             "step into", "step over", "step out"}
```

Other commands have no effect on the control mode.

```
isBreakingCommand(x) =
   not(isRunningCommand(x))  and  x != "exit" and x != "start"
```

As mentioned above, in the control model, handling of commands is a skip operation. This rule is refined in the object model.

```
doCommand(x) = skip
```

Callbacks The debugger can handle callbacks or responses from the runtime only in the `Run` mode. The value of the monitored function `response` is a callback message from the runtime notifying the debugger about a runtime event. In the debugger, each event is classified either as a *stopping* event or as a *non-stopping* event.

```
handleResponses      = do onStoppingEvent
                          onNonStoppingEvent
                          response := "nothing"

onStoppingEvent    =
  if isStoppingEvent(response) then do dbgMode:= Break?
                                       doCallback(response)

onNonStoppingEvent =
  if not(isStoppingEvent(response)) then doCallback(response)
```

Breakpoint hit events, step complete events, and process exit events are always stopping events. Initially `isStoppingEvent` is the following unary relation (unary Boolean function).

```
isStoppingEvent(x) = x == "step completed" or x == "breakpoint hit" or
                     x == "process exited"
```

However, the relation is dynamic and may evolve during a run as a result of a specific user command or a specific runtime event.

In the control model the actual handling of callbacks is a skip operation. This rule is refined in the object model.

```
doCallback(x) = skip
```

Pending Events In `Break?` mode a stopping event has happened and the debugger should hand the control over to the user. This happens only if there are no *pending* events in the runtime. Otherwise the control goes back to the runtime and the debugger continues the current process

```
handlePendingEvent = do onPendingEvent
                        onNoPendingEvent

onPendingEvent     =  if isEventPending then do dbgMode := Run
                                                isEventPending := False
                                                doCommand("continue")

onNoPendingEvent   =  if not(isEventPending) then do dbgMode := Break
```

The boolean function `isEventPending` is monitored by the runtime.
The control model is summarized by a state diagram in Figure 3.

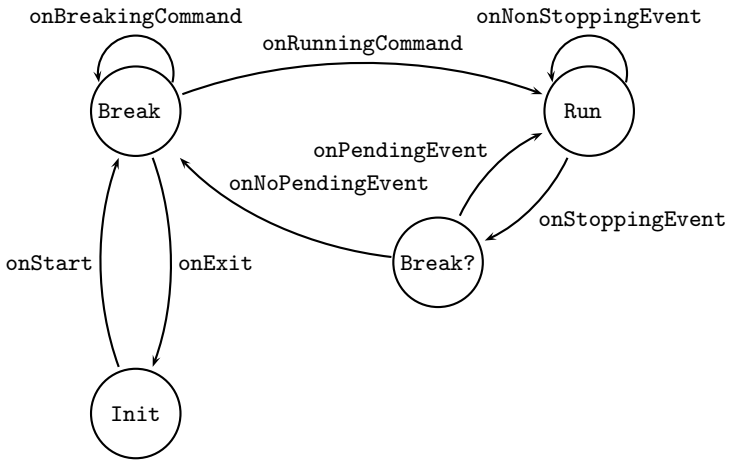

Fig. 3. Control model of the debugger.

3.2 A Wizard-of-Oz Experiment

We want to explore the behavior of the model on a given set of user scenarios, without having access to the real runtime, and possibly expose contradictions in the model. Since we reverse-engineered the debugger, we cannot claim that our model is truly faithful to it. In fact, any error that might show up may very well have sneaked into the model without being in the debugger. The point we want to make is that this form of testing *would* be useful if used during the design phase.

Since the actual runtime is missing, we will ask a "wizard" to play its role. Table 1 shows a run, with row i indicating the part of the state that has changed after the i'th ASM step of the model.

	dbgMode	command	response	isEventPending
0:	Init	start		
1:	Break	bp hello.cpp:7		
2:	Break	run hello.exe		
3:	Run		created process	
4:	Run		loaded module	
5:	Run		created thread	
6:	Run		hit breakpoint	
7:	Break?			True
8:	Run		loaded class	
9:	Run		...	

Table 1. A run of the control model.

The run shows that after we hit a breakpoint there is a pending event in the runtime. According to the model, the current process is continued and the control is passed to the runtime. It turns out that the pending event was a non-stopping event (class was loaded). Obviously, this behaviour contradicts the expected consequence of reaching a breakpoint, namely that the user should get the control. At this point we have two options to solve the problem: if we can constrain the runtime to meet the restriction that only stopping events can cause the `isEventPending` flag to become true, then the current model is correct; otherwise we have to modify the model. We choose the latter, see Figure 4.

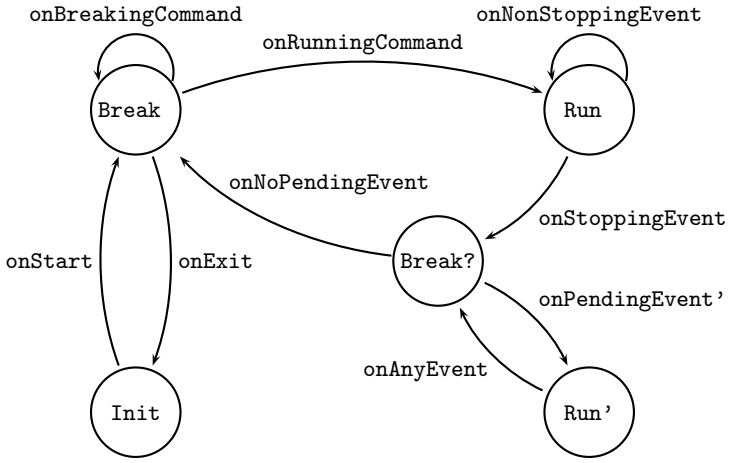

Fig. 4. New control model of the debugger.

4 Object Model

The static and the dynamic structures of the runtime architecture are reflected in the object model with just enough detail so that one can model features that deal with execution control, breakpoints, and stepping, and explore their interaction.

Let us consider some examples. The first example is the refinement of the user command that starts the debugger. The next two examples specify what happens in the object model when a module is loaded or unloaded in the runtime. Recall that all those rules are just `skip` rules in the control model.

4.1 User commands

Starting the debugger

```
doCommand("start") = do in-sequence
  coInitialize
  shell := newShell(services    = newServicesInterface(...),
                    callback    = newCallbackInterface(...),
                    modules     = {},
                    breakpoints = {},
                    debugee     = undef)
  shell.services.setCallback(shell.callback)
```

The first rule initializes the COM environment. The second rule creates a new debugger shell with a pointer to the services and a new callback. The third rule invokes the setCallback method of the debug services with the new callback interface pointer as argument, thus providing the services access to the client's callback methods.

4.2 Callbacks

Loading of modules

```
doCallback(<"load module",mod>) = do
  shell.modules := shell.modules ++ {mod}
  do in-sequence
     forall bp in shell.breakpoints do bp.bind(mod)
     shell.debugee.continue()
```

The new module is recorded in the shell. The shell attempts to bind each of its set of breakpoints to that module. Once all new bindings have been made (if any), the debugee is continued through an external call.

We have omitted the `bind` rule that checks if the location that the breakpoint refers to indeed exists in the module. If the location exists, a real breakpoint is created at that location through an external call to the services; otherwise, nothing is done.

Unloading of modules

```
doCallback (<"unload module",mod>) = do
  shell.modules := shell.modules \ {mod}
  do in-sequence
    forall bp in shell.breakpoints do bp.unbind(mod)
    shell.debugee.continue()
```

The effect of unloading a module is to update the shell and to remove all the breakpoints from that module.

4.3 Some comments

There are a couple of things worth noticing about the above rules. First, the design decision to bind a breakpoint that may already be bound implies that if there is a breakpoint referring to a location that exists in two or more distinct modules, then the breakpoint is associated to all of them. Second, all breakpoints are handled simultaneously; there are no ordering constraints between them. This is a typical situation: in the actual (C++) code there is a linearly ordered structure maintaining the elements that are then processed sequentially in the order determined by the structure.

When constructing the object model we detected a mismatch between the way loading and unloading of module callbacks was implemented. Namely, although loading followed the specification above, during unloading of a given module, if a breakpoint was bound to this module, then that breakpoint was not only removed from this module but from *all other* modules as well. We did not discover this when studying the code, but only when constructing the object model. In fact, it is hard to see it from the code, but readily apparent from the model.

5 Ground Model

The ground model has the same core functionality as the debugger and can be executed as an AsmHugs program that communicates with the runtime by using the H/Direct generated glue code. It gives a more detailed view of the run time structure than the object model. All the user commands, such as inspection of specific object fields, are modeled here. The ground model is fully described in the technical report [3].

6 Conclusion

The case study illustrates some particular ways that ASMs can help the system designer:

– *Concisely describe complex systems.* The debugger involves several asynchronous agents and involves about 50 interfaces and about 250 methods. The size of the debugger is about 30K lines of C++ code. The size of the ASM specification (which can be run, though not as fast, and which provides essentially the same functionality) is only 4K lines. Due to built-in parallelism, the runs of our ASM are shallow. In fact it takes only bounded many steps (less than 10) to process one user command in contrast to the C++ code which may require unbounded many steps for the purpose (the number of steps may depend on the program being debugged). The parallelization was obtained, e.g., by abstraction from irrelevant sequentialization in the C++ code (see Section 4.3).
– *Abstract away the environment as needed.* We could easily separate the models, but still formally tie them via refinements, without losing executability.
– *Interactively explore the design on all abstraction levels.* It is very difficult to detect at the source code level such high level bugs as the ones that we detected with relative ease with the Wizard-of-Oz experiment. The object model enabled us to detect inconsistencies in the callback management.

To repeat, our goal is to provide a rigorous method, clear notation and convenient tool support for high-level system design and analysis. Our main tool will execute ASMs (and will help you to write ASMs). Several issues that arose during the case study have influenced the design of that tool.

– We found the parallel synchronous construct `forall` very useful. Similar conclusions were drawn from the Falko project [4].
– Set and list comprehensions turned out to be very convenient.
– We found object oriented notation useful to structure specs, to improve their readability, and to link their execution to the object-oriented programming paradigm.
– We realized that in order for ASMs to be useful in Microsoft (or indeed anywhere COM is used), ASM models must achieve full COM interoperability.

The first and the third point are illustrated by the examples in Section 4.2.

Acknowledgments

We thank Sigbjorn Finne for helping us with H/Direct during the course of this work.

References

1. AsmGofer. `http://www.tydo.de/AsmGofer/`.
2. Hugs98. `http://www.haskell.org/hugs/`.
3. Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using ASMs at Microsoft: A case study. Technical report, Microsoft Research, Redmond, USA, 2000.

4. Egon Börger, Peter Päppinghaus, and Joachim Schmid. Report on a practical application of ASMs in software design. In *This Volume*.
5. Don Box. *Essential COM*. Addison-Wesley, Reading, MA, 1998.
6. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proc ACM Sigplan International Conference on Functional Programming (ICFP'98), Baltimore*, pages 153–162, 1998.
7. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Proc ACM Sigplan International Conference on Functional Programming (ICFP'99), Paris, France*, 1999.
8. Foundations of Software Engineering, Microsoft Research. AsmHugs. `http://www.research.microsoft.com/foundations/`.

to appear in: Y.Gurevich, M.Odersky, P.Kutter, L.Thiele (Eds): "International Workshop on Abstract State Machines ASM'2000", Springer LNCS, 2000