Two new ways to do …

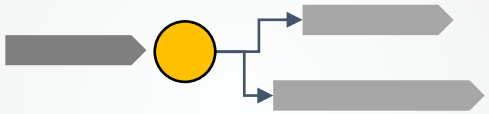
Automatic Checkpoint Support in the Device Modeling Language (DML)

Jakob Engblom, jakob.engblom@intel.com

Erik Carstensen, erik.carstensen@intel.com

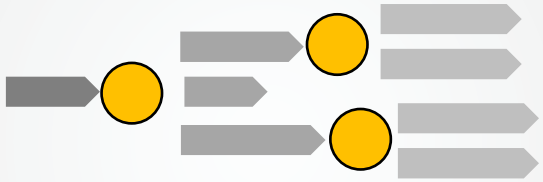
Checkpointing is very Useful in Virtual Platforms

"Save the boot/setup"



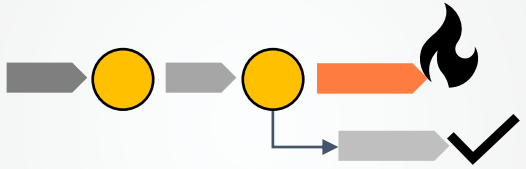
Avoid redoing work, save booted & configured system for quicker start test cases & other work

Parallelize test execution



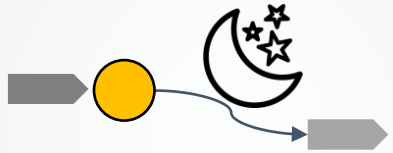
Save checkpoints after important stages. Open the same checkpoint for multiple parallel runs with different inputs.

Undo target actions



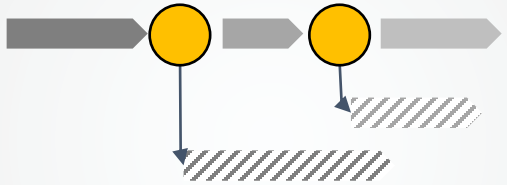
Get back to a previous good state after mistakes that break the target setup

Save for the day



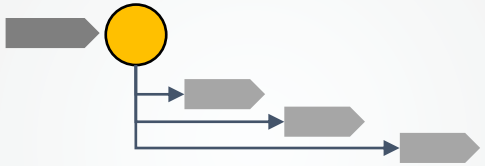
Save work, shut down simulation, continue the next day

Gear shifting/state transfer



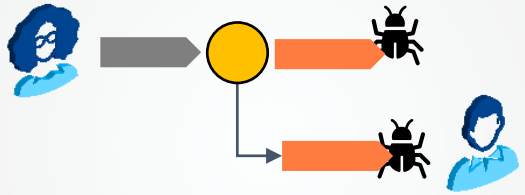
Save state from fast VP, to start performance/power/thermal simulations from a certain software state

Fuzzing



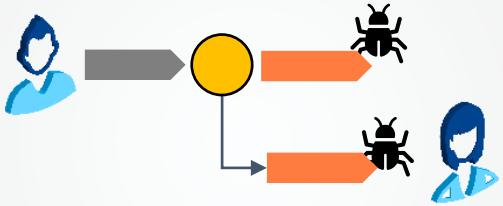
Repeatedly restore the same checkpoint, send different inputs to code under fuzzing – to save setup time

Report software bugs



Save state of system when bug hits, transfer to SW developer for analysis

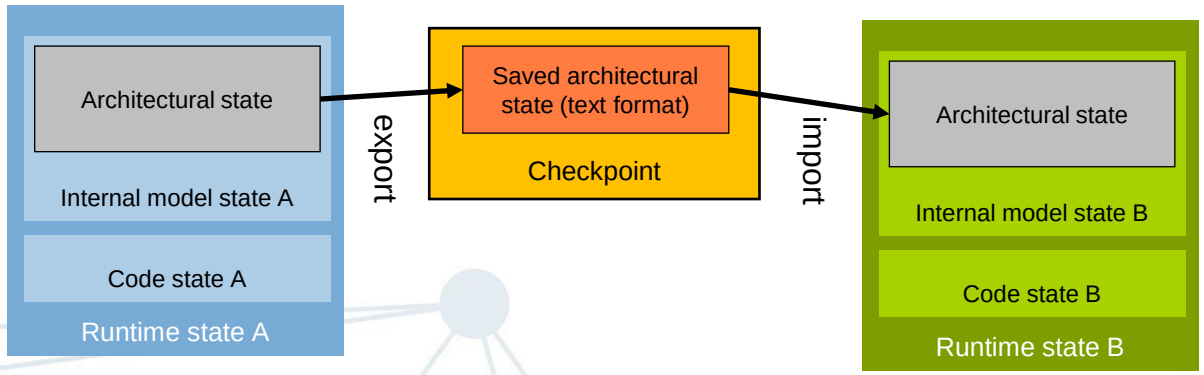
Report model bugs



Provide HW + SW state when a model bug hits to model developer for analysis

Checkpointing is Hard to Implement Right

- Checkpointing requires care and effort in how the model is written
  - Keep all state on the heap (not on the stack)
  - Provide code to export and import model state
  - Model must handle importing state post-setup
  - Manage pointers to other objects
    - (DML and the Simics® simulator uses names of objects)
  - Manage state attached to events (reposting after import)



- The Device Modeling Language (DML) is designed to make checkpointing automatic and invisible
  - Core language constructs create heap-only state
  - Attributes to handle configuration and saved state
  - Still… there are cases that require manual coding, this work is part of an ongoing process to reduce them

- This talk: two new checkpointable features
  - Saving **internal model variables**
  - Simplify **event posting**

New Automatically Checkpointed Features

- **Model-internal state in (complex) variables**
  - Data stored in a variable
    (and not in a register or explicit configuration attribute)
    - Typically: internal data structures in structs and/or arrays
  - Checkpointing required adding attributes and manually coding export/import functions
    - Burdensome to do (added at least 2 methods + 1 declaration)
    - Easy to forget
    - = Checkpoint can be silently broken even for the best-intentioned modelers
- Solution: introducing the **`saved`** keyword

- **Events (with data)**
  - DML **`after`** statements make event-posting easy
  - Used to support only events with no data
    - Passing arguments to an event callback required declaring an event object
    - Adding at least 4 methods of mostly boiler-plate code
    - Forced data to be encapsulated in structs
- Solution: **`after`** statements now support arguments
  - With automatic checkpointing
  - No structs needed to handle multiple arguments

Common Core: State Import and Export

- **Automatic generation of state export and import code**
  - Both `saved` and `after` require the same mechanism
- The Device Modeling Language (DML) compiler backend generates the necessary code
  - Details of serialization are left to the compiler
  - Provides the option of **simulator independence**
    - Code expresses the **intention** of the programmer
  - Cannot really be done using a library in a general-purpose programming language
- No impact on run-time performance

- **Supported (serializable)** types in DML:
  - **Primitive values**: Booleans, integers, floating point
  - **Structs** of serializable types
  - **Fixed-length arrays** of serializable types
- I.e., complete type must be known at compile time
  - DML encourages compile-time specialization
  - In most cases, arrays sizes are known at compile time (even if their length is a parameter in the model code)
- **Excluded** types:
  - **Pointers** – unknown ownership and rules for the pointed-to data
  - **Dynamically allocated arrays** (which are really pointers)
  - **Local variable-length arrays** in functions

Code: Saved vs Manual Save

```
// Device-internal data structure
typedef struct {
  bool full;
  int head;
  int tail;
  uint32 values[circular_capacity];   // constant known at compile-time
} circular_buffer_t;


// Solution with saved
// -- Generates an attribute called "cbuf", see below
//
saved circular_buffer_t cbuf;
```

Code is shorter and easier to read. And much much easier to write.

Intel® Simics® command-line interface

```
# Structured and understandable value
simics> device->cbuf
[FALSE, 2, 0, [161, 162, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
// Manual coding needed when NOT using saved, to produce the same result
session circular_buffer_t cbuf;  // Session variable only lives within a simulation run
attribute a_cbuf {
  param type = "[bii[i{10}]]";   // Format of attribute data
  // Export code, in the get() function
  method get() -> (attr_value_t) {
    local attr_value_t a;
    local attr_value_t l;
    local int i;
    a = SIM_alloc_attr_list(4);
    SIM_attr_list_set_item(&a,0, SIM_make_attr_boolean(cbuf.full));
    SIM_attr_list_set_item(&a,1, SIM_make_attr_int64(cbuf.head));
    SIM_attr_list_set_item(&a,2, SIM_make_attr_int64(cbuf.tail));
    l = SIM_alloc_attr_list(10);
    for(i=0;i<10;i++) {
      SIM_attr_list_set_item(&l,i, SIM_make_attr_uint64(cbuf.values[i]));
    }
    SIM_attr_list_set_item(&a,3, l);
    return a;
  }
  // Import code, in the set() function
  method set(attr_value_t a) throws {
    local attr_value_t l;
    local int i;
    cbuf.full = SIM_attr_boolean(SIM_attr_list_item(a,0));
    cbuf.head = SIM_attr_integer(SIM_attr_list_item(a,1));
    cbuf.tail = SIM_attr_integer(SIM_attr_list_item(a,2));
    l = SIM_attr_list_item(a,3);
    for(i=0;i<10;i++) {
      cbuf.values[i] = SIM_attr_integer(SIM_attr_list_item(l,i));
    }
  }
```

Code: After vs Manual Event

```
// Simplest solution: after + using multiple arguments to the function
method WorkMethod(int operation, uint64 data_loc, uint32 data_len) {…}
after 2 s: WorkMethod( 1, 0xff000000,  0x100 );
```

This is the best way to express the operation

```
// For manual data management, as required when using events
// - Define a struct to hold the method arguments
typedef struct {
  int    operation;
  uint64 data_loc;
  uint32 data_len;
} worker_args_t;

method WorkMethodBis(worker_args_t a) {…}

method some_method() {
  local worker_args_t a = {.operation = 2,
                           .data_loc  = 0xff000000,
                           .data_len  = 0x100};
  after 2 s: WorkMethodBis( a );
}
```

Packing arguments into a struct requires more code…

In the checkpoint file

```
OBJECT de.clock.vtime.cycles TYPE cycle-counter {
…
events: ((de.dev,"WorkMethod",(("arguments",((2,0xff000000,256))),…
}
```

```
// Using a manually coded to event to achieve the same - note manual memory management
event EventWithArgs is (custom_time_event) {
  method event(void *data) {
    local worker_args_t *a_data = cast(data,worker_args_t*);
    WorkMethodBis(*a_data);
    delete a_data;
  }
  // Export to checkpoint, as a pile of binary data
  method get_event_info(void *data) -> (attr_value_t) {
    local attr_value_t info;
    local worker_args_t *a_data = cast(data,worker_args_t*);
    info = SIM_make_attr_data(sizeoftype(worker_args_t),a_data);
    return info;
  }
  // Import from checkpoint
  method set_event_info(attr_value_t info) -> (void *) {
    local void *data;
    local worker_args_t * buf = new worker_args_t;
    memcpy(buf, SIM_attr_data(info), sizeoftype(worker_args_t));
    data = cast(buf, void *);
    return data;
  }
  // Destroy data, in case an event is cancelled before it triggers
  method destroy(void* data) {
    local worker_args_t *a_data = cast(data,worker_args_t*);
    delete a_data;
  }
}

// Posting the event to trigger after 1 second
local worker_args_t* b = new worker_args_t; // heap-allocated variable!
b->operation = 3;
b->data_loc  = 0xaa00bb00;
b->data_len  = 0x100;
EventWithArgs.post(1, cast(b, void*));
```

Using an event to save the struct requires a ton of manual code: 4 additional methods

Manual memory allocation and deallocation for the argument structure

Question: Integration of C/C++ Code?

- Integrating code written in C or C++ is supported by DML
- Checkpointability depends on the integrated code
- Simple case:
  - Keep all model state in the DML code
  - = C/C++ used to compute results within a single simulation step
- Complex case:
  - = C/C++ code manages its own state
  - Requires adding explicit export and import code

Summary

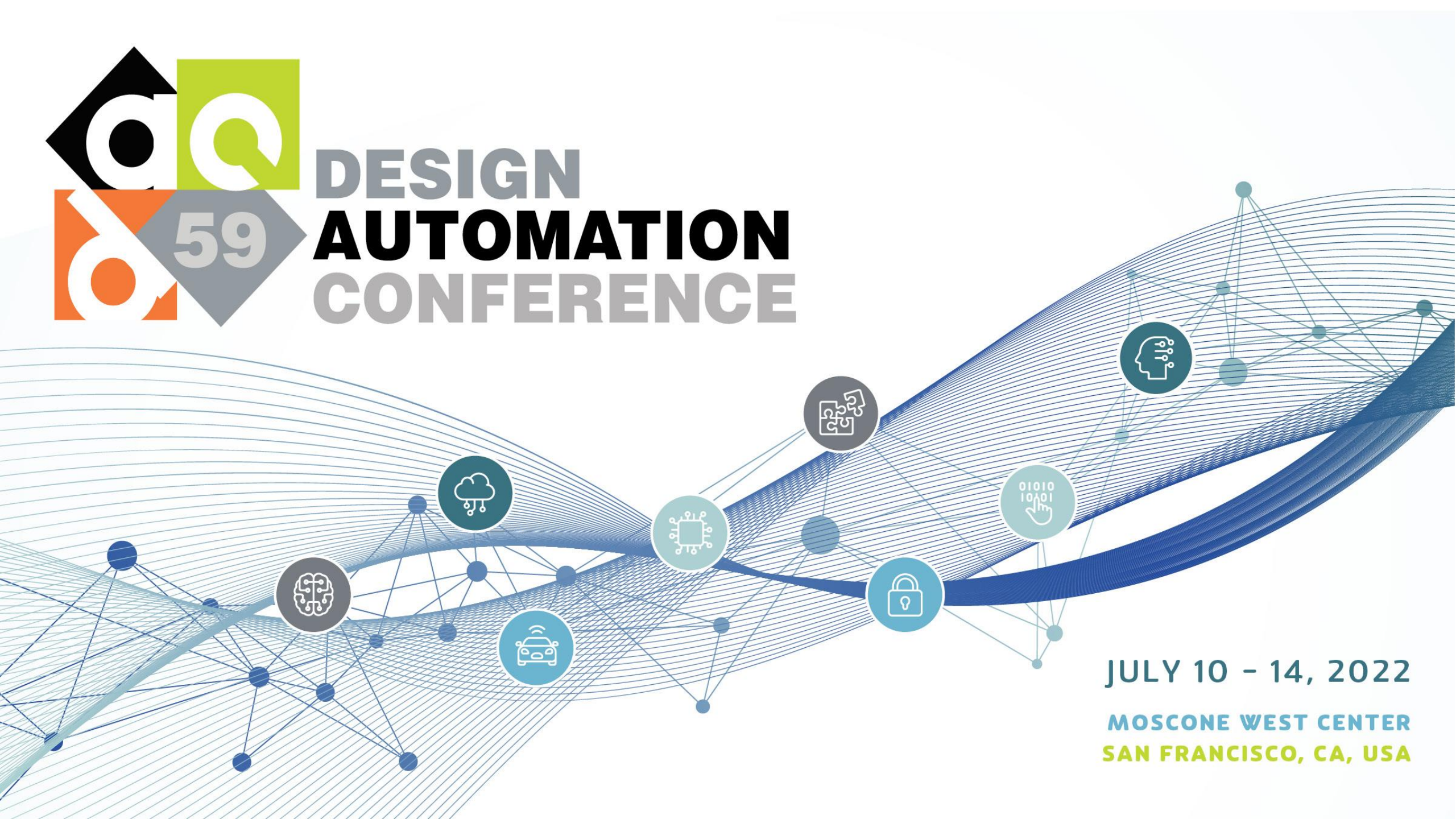
- The Device Modeling Language (DML) has added support for automatic checkpointing of saved variables and after arguments
  - (Much) shorter and clearer code
  - Broadens the set of models that can be checkpointed automatically
  - Easier to support other simulator environments by avoiding raw API calls
  - No runtime impact

Compiler vs Library

- This is a good example of the power of using a custom compiler
  - Instead of implementing modeling using a library in a general-purpose language
- Domain-specific languages backed by a compile-time code generator:
  - Raises the level of expression
  - Increases portability of code
  - Allows more compact source code
  - Allows functionality that is not possible to implement using code-level features (macros, templates, classes, etc.)
  - Allows the compiler to add new functionality to existing models

It’s all in Open-Source

- The Device Modeling Language (DML) and its compiler is open source:
  - https://github.com/intel/device-modeling-language
  - To run models, you can use the public release of the Intel® Simics® Simulator: https://developer.intel.com/isim
- Supporting other simulator backends?
  - Not currently implemented
  - This work is a step towards removing Simics simulator APIs from DML source code


59
DESIGN
AUTOMATION
CONFERENCE
JULY 10 - 14, 2022
MOSCONE WEST CENTER
SAN FRANCISCO, CA, USA