Verification Patterns in Addition to RVM

Carl Cavanagh

Christopher Sine

Lee Warner

Sun Microsystems, Inc.

Background

- Started 3 new ASIC developments 4 years ago.
  - Schedule and resources were tight.
- In house verification framework(s) support and maintenance was in question.
- Decision made to standardize on RVM
  - Good documentation, support and maintenance.
- RVM provided a running start
  - Verification strategies, architecture guidelines and set of extensible verification environment components.

Background

- However, RVM left us in the dark with respect to a number of environment aspects:
  - Reusing configuration facilities across full chip and block level environments.
  - How to manage global environment parameters.
  - How to encapsulate design configurations.
  - How to pass side band verification information.
  - Checker/scoreboard architecture.
  - Error injector control.
- This presentation and the associated paper document some of the solutions we developed to build on RVM.

Agenda

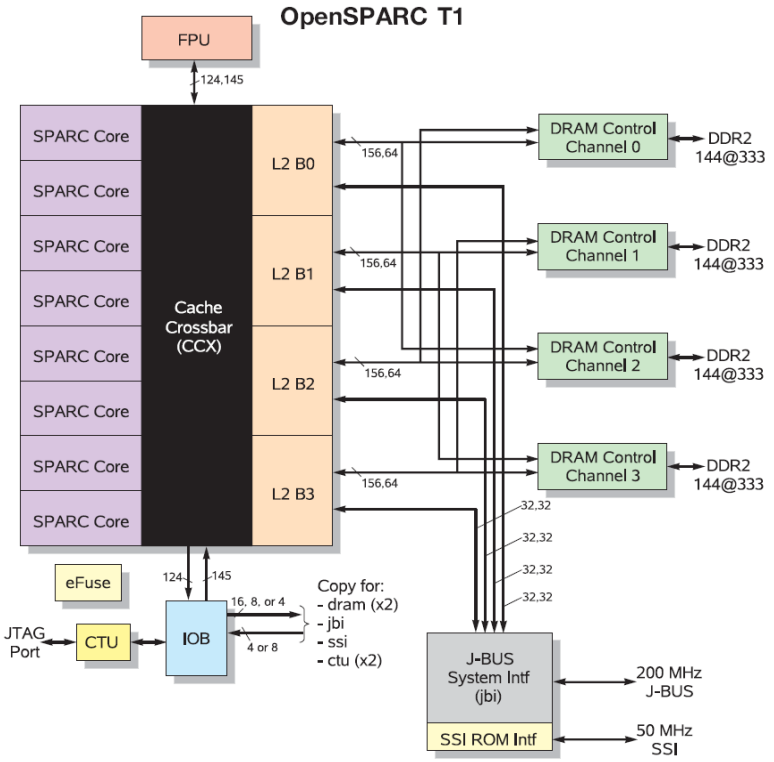
- The following topics will be addressed during this presentation:
  - Verification Reuse
  - Example Introduction
  - Parameter Containers
  - Configuration Descriptors
  - Meta Data
  - Conclusion

Verification Reuse

- Verification component reuse was extremely important in our project.
  - Needed to share generators, transactors, checkers, configuration routines between chip environments and block level environments.
- Useful software architecture guidelines:
  - Decouple objects that interact.
  - Favour composition over inheritance.
  - Program to an interface not an implementation
  - Open Closed Principle.

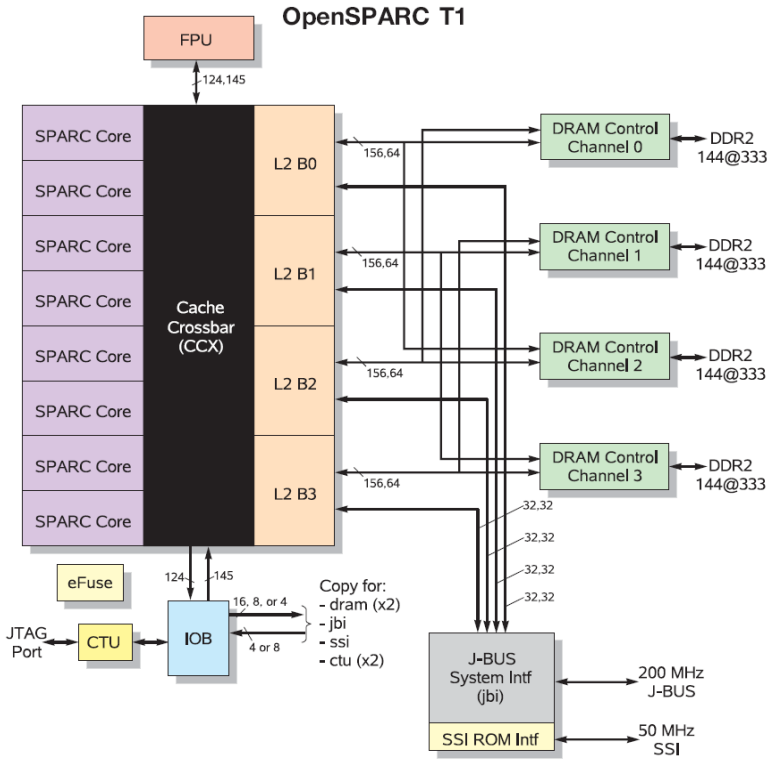
Example Introduction

- This presentation will use the OpenSPARC T1 CMT Mirco Processor, to help demonstrate the concepts being outlined.
- It should be noted that the authors of this presentation and the associated paper have no affiliation with the OpenSPARC project or developing the UltraSPARC T1/T2 family of processors (basically we're not experts... so please accept our apology in advance if we get something wrong!!)



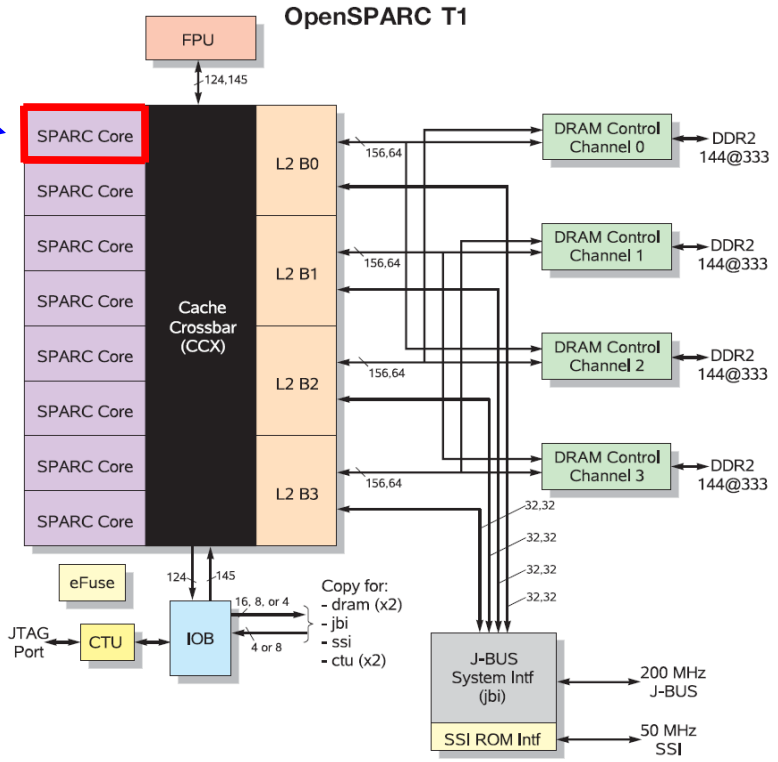
Example Introduction

- To verify the OpenSPARC T1 CMT Mirco Processor we would most likely break it up into a number of stand alone test environments (SATs).



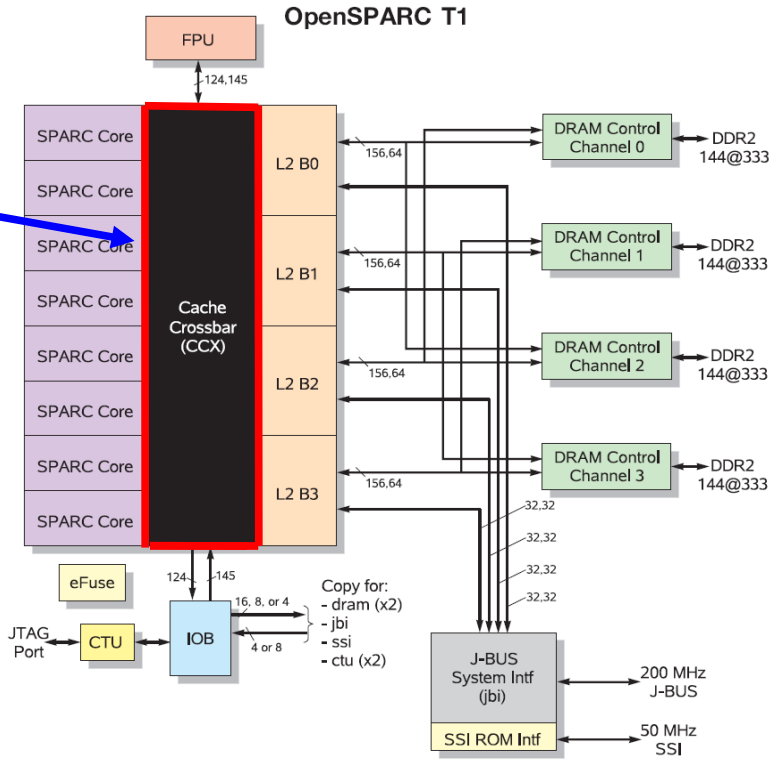
Example Introduction

- To verify the OpenSPARC T1 CMT Mirco Processor we would most likely break it up into a number of stand alone test environments (SATs).
  - A Sparc Core SAT



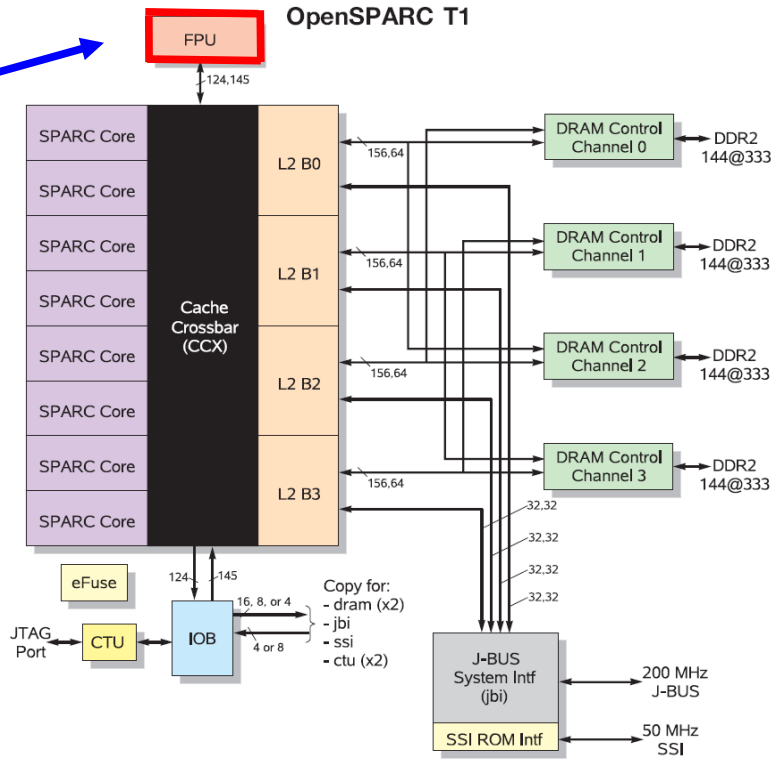
Example Introduction

- To verify the OpenSPARC T1 CMT Mirco Processor we would most likely break it up into a number of stand alone test environments (SATs).
  - A Sparc Core SAT
  - A CCX SAT



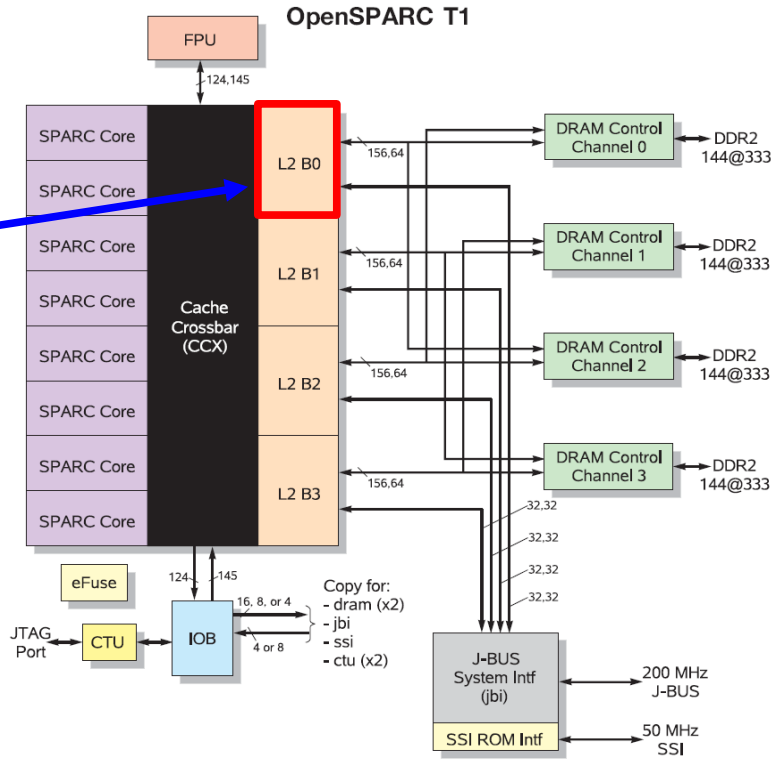
Example Introduction

- To verify the OpenSPARC T1 CMT Mirco Processor we would most likely break it up into a number of stand alone test environments (SATs).
  - A Sparc Core SAT
  - A CCX SAT
  - A FPU SAT



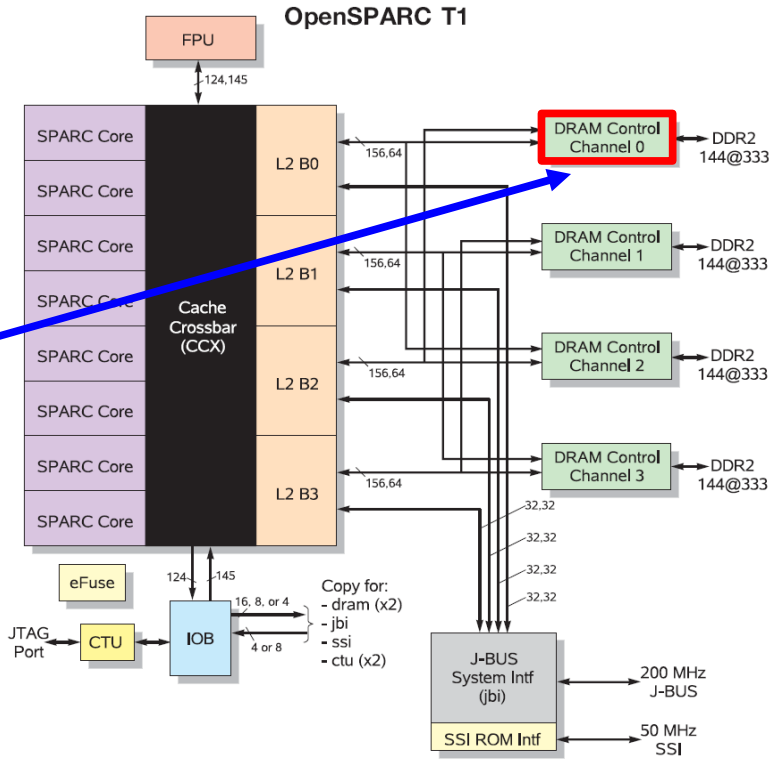
Example Introduction

- To verify the OpenSPARC T1 CMT Mirco Processor we would most likely break it up into a number of stand alone test environments (SATs).
  - A Sparc Core SAT
  - A CCX SAT
  - A FPU SAT
  - A L2 SAT



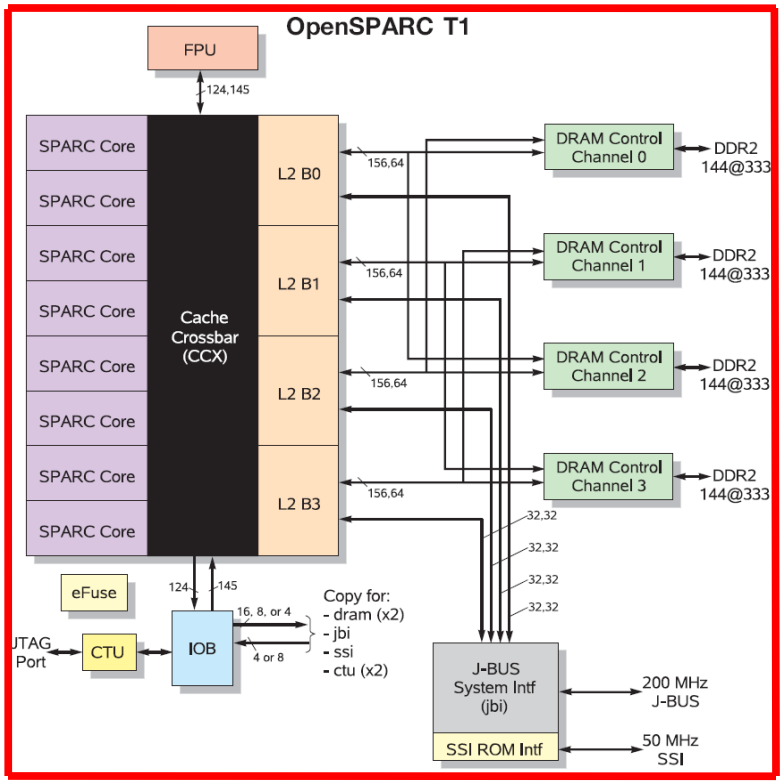
Example Introduction

- To verify the OpenSPARC T1 CMT Mirco Processor we would most likely break it up into a number of stand alone test environments (SATs).
  - A Sparc Core SAT
  - A CCX SAT
  - A FPU SAT
  - A L2 SAT
  - A DRAM Control SAT



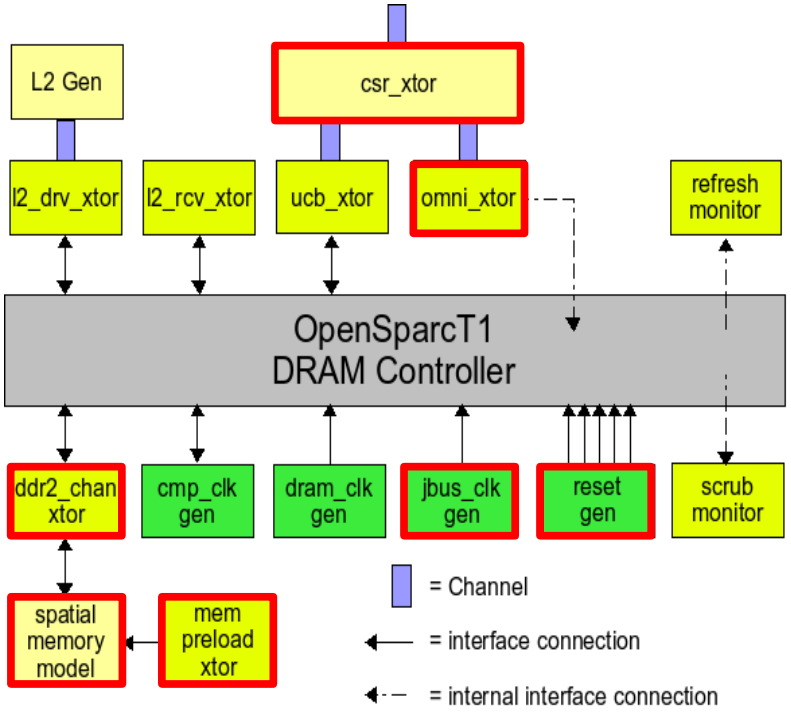
Example Introduction

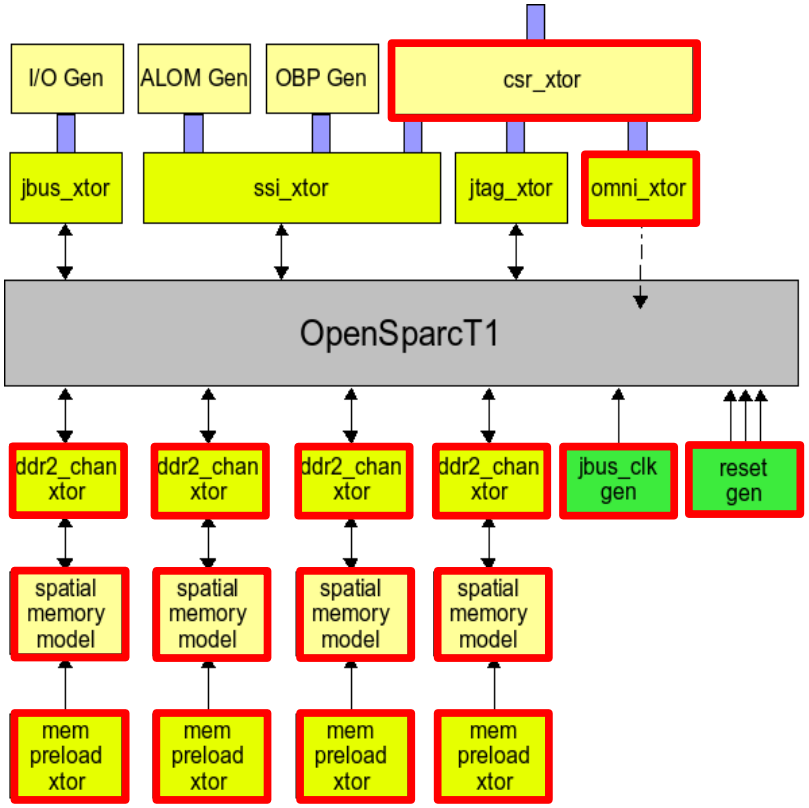
- To verify the OpenSPARC T1 CMT Mirco Processor we would most likely break it up into a number of stand alone test environments (SATs).
  - A Sparc Core SAT
  - A CCX SAT
  - A FPU SAT
  - A L2 SAT
  - A DRAM Control SAT
  - We would also want to develop a Fullchip (FC) test environment as well.



Parameter Containers

- The Problem
  - Each of the OpenSPARC T1 verification environments will reuse many of the same components. For example:





Parameter Containers

- The Problem
  - Since the environments are sharing common generators, transactors and checkers they will also be sharing common configurations and parameters e.g.
    - ➢Which cores are enabled?
    - ➢The memory configuration?
  - The management and sharing of parameters that describe the environment context and configuration can get complicated.

Parameter Containers

- The Problem
  - RVM promotes the following:
    - ➢The environment's gen_cfg method should be used to randomize the context of the environment.
    - ➢The device under test should be configured by the environment's cfg_dut_t method.

Parameter Containers

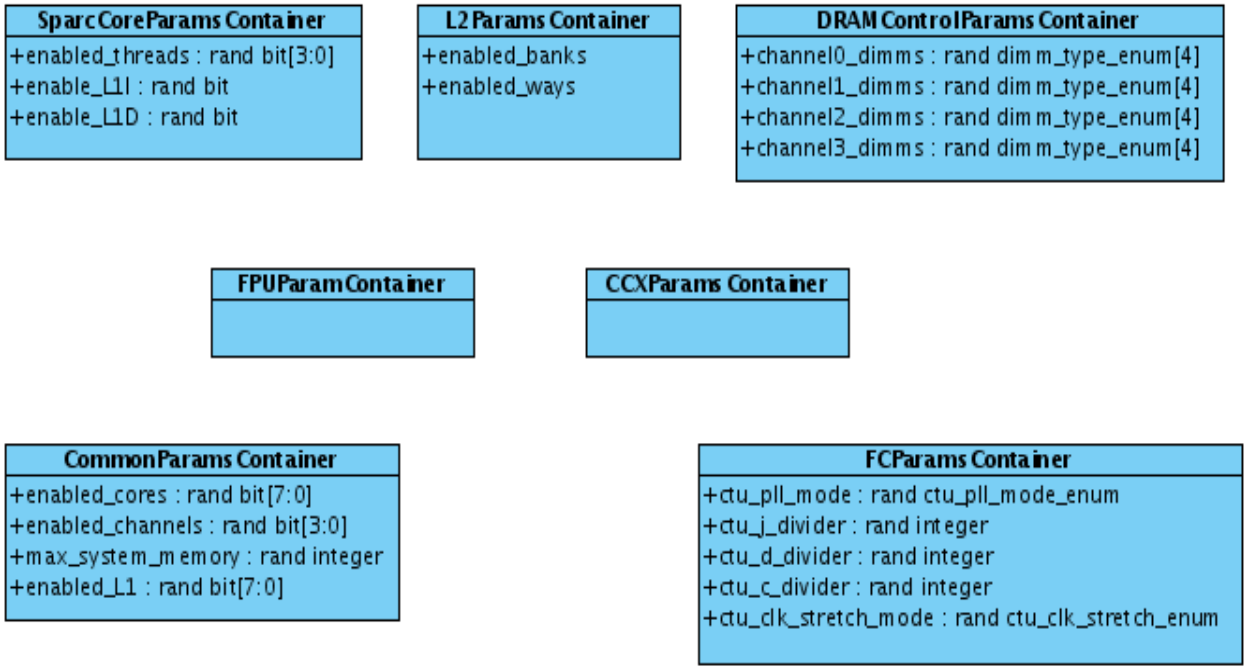
- The Problem
  - RVM promotes the following:
    - ➢The environment's gen_cfg method should be used to randomize the context of the environment.
    - ➢The device under test should be configured by the environment's cfg_dut_t method.
  - However,
    - ➢These methods cannot be shared between multiple environments.
    - ➢The “common parameters” and configuration are tightly coupled to the environment itself.

Parameter Containers

- The Natural Solution
  - Follow RVM's lead and implement a configuration object for each of the rvm_env derivatives.
  - But,
    - ➢This still couples the configuration to a given environment.
    - ➢The same configuration object attributes will need to be replicated for each environment.
    - ➢Each environment will need to map these attributes into the verification component configuration objects.

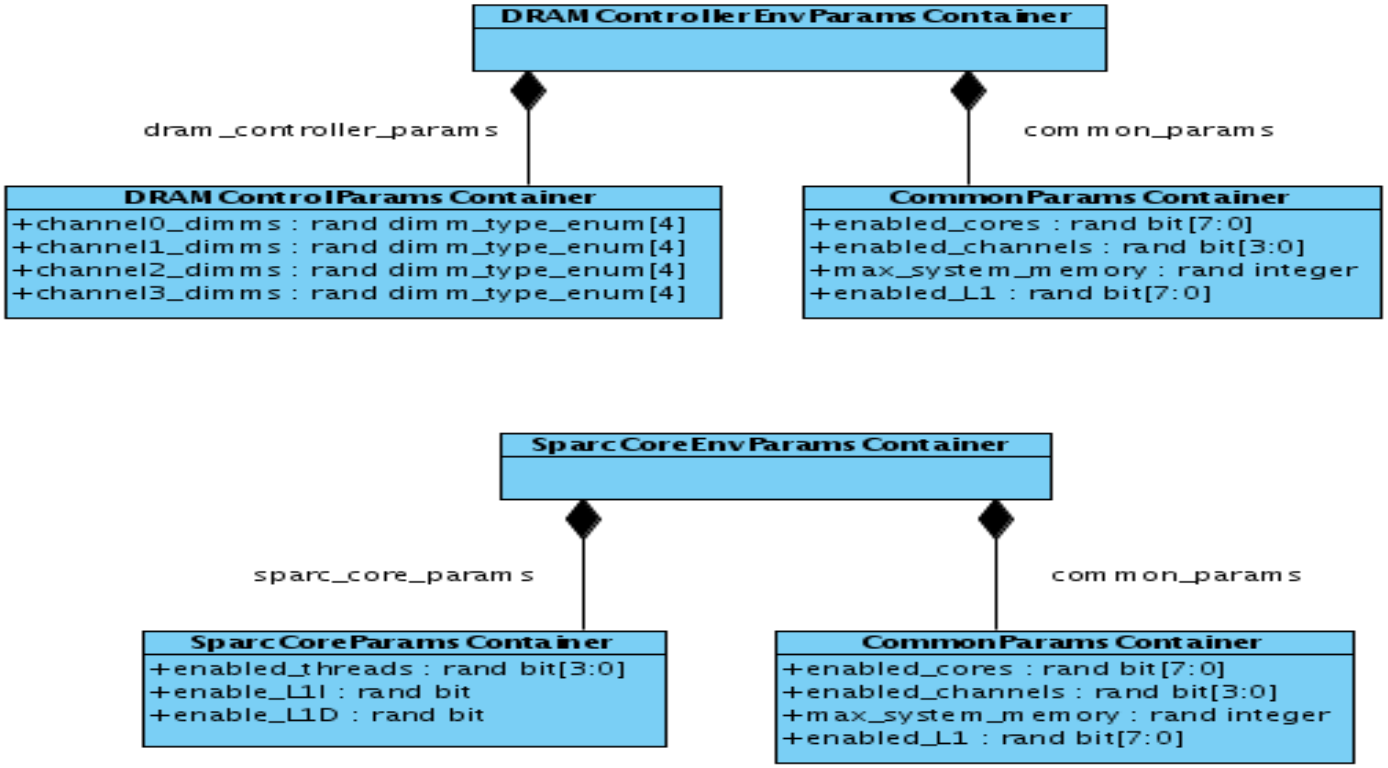
Parameter Containers

- Composed Parameter Container Solution
  - Implement randomizable configuration objects that are associated with a given SAT in general rather than just the rvm_env object.



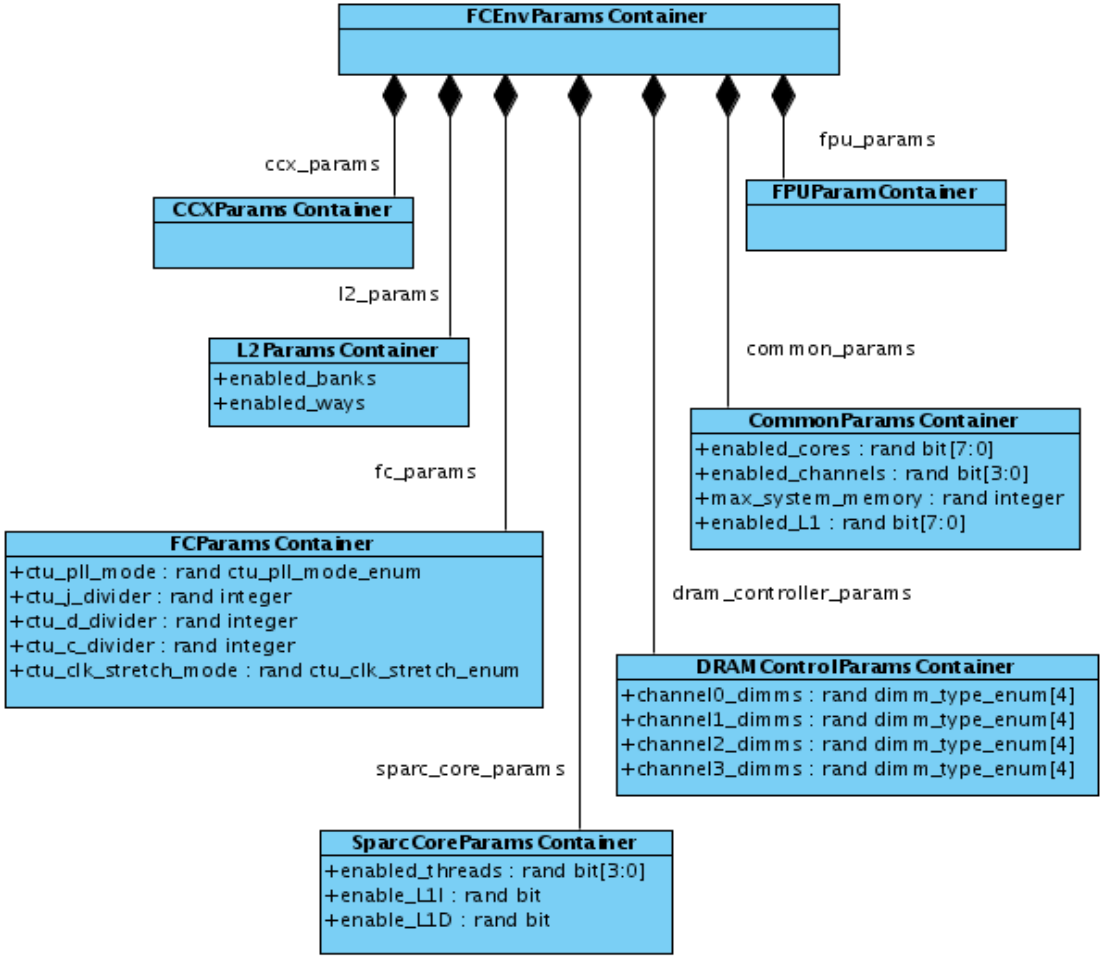
Parameter Containers

- Composed Parameter Container Solution
  - Compose a container for the environment and its components that includes all the parameters it needs.



Parameter Containers

- Composed Parameter Container Solution



Parameter Containers

- Composed Parameter Container Solution
  - Given the following:
    - The instance names of each of the composed parameter containers is always the same.
    - The instance name of the container in the environment is always the same.
  - A verification component will always know where to find a given parameter in any environment.
  - Common constraints can be declared that can be applied across environments.
  - All the parameters are randomized when the top most container is randomized.

Configuration Descriptors

- The Problem
  - Most devices under test and their environments will require some form of configuration.
    - Programming registers.
    - Asserting control signals.
    - Manipulating environment component behaviour.

Configuration Descriptors

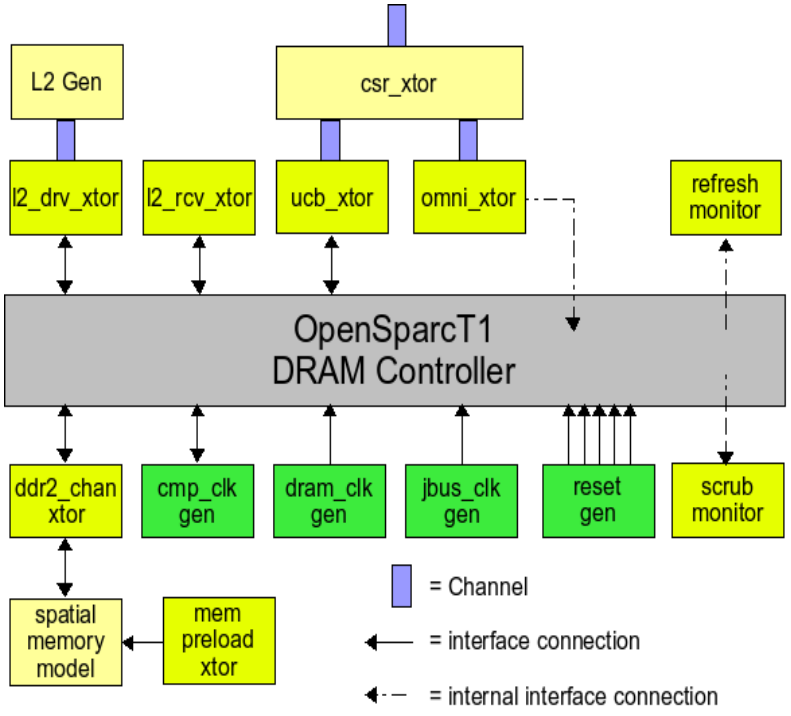
- The Problem
  - Most devices under test and their environments will require some form of configuration.
    - Programming registers.
    - Asserting control signals.
    - Manipulating environment component behaviour.
  - RVM describes that configuration variables should be randomized in the environment's gen_cfg method, and device configuration performed in its dut_cfg_t method.

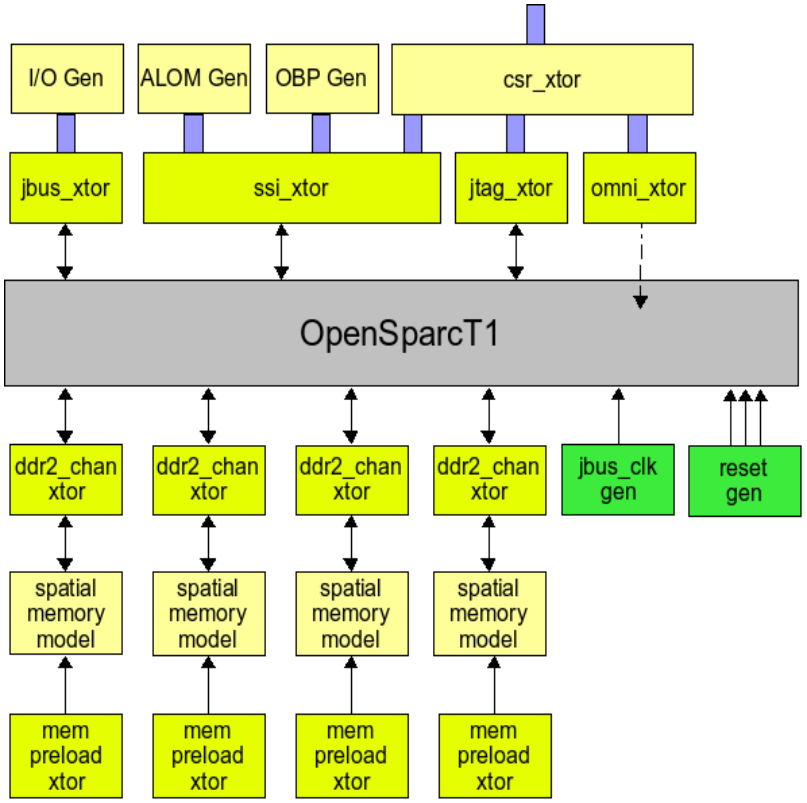
Configuration Descriptors

- The Problem
  - Most devices under test and their environments will require some form of configuration.
    - Programming registers.
    - Asserting control signals.
    - Manipulating environment component behaviour.
  - RVM describes that configuration variables should be randomized in the environment's gen_cfg method, and device configuration performed in its dut_cfg_t method.
  - This potentially leads to coding the configuration routines directly into this environment methods, thus coupling them to a single verification environment.

Configuration Descriptors

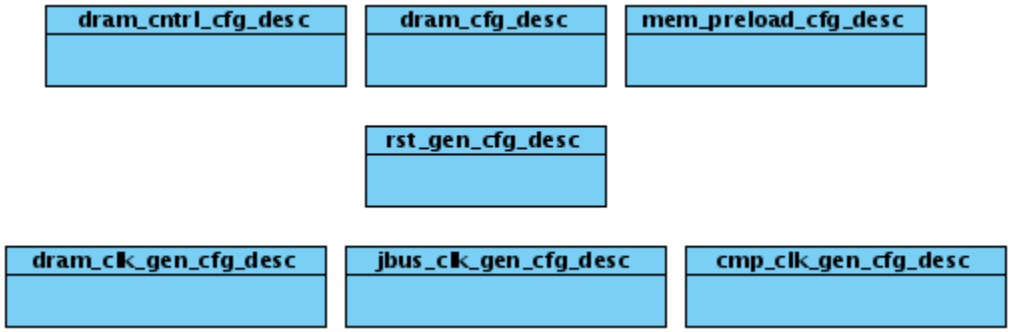
- The Problem
  - If we want to reuse the configuration routines between environments we need to decouple them.





Configuration Descriptors

- The Composition Approach
  - The natural solution is to encapsulate the configuration code into configuration descriptor (cfg_desc) objects that can be reused by any environment.



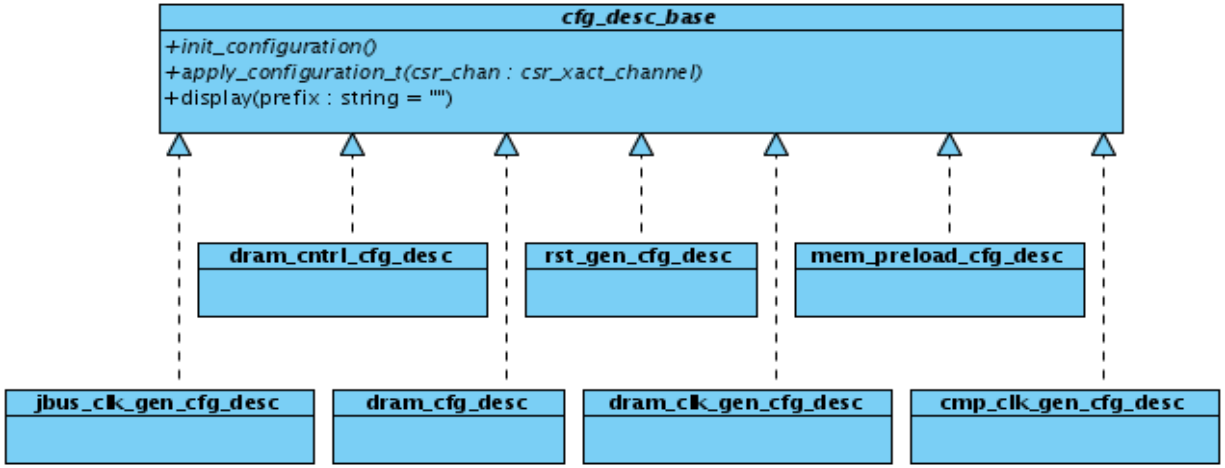
  - These cfg_desc's decouple the configuration implementation from the environment.
  - However, the environment is now coupled to their individual implementations.

Configuration Descriptors

- The Composition Approach
  - To resolve this we can employ the following design principle “program to an interface not an implementation”.

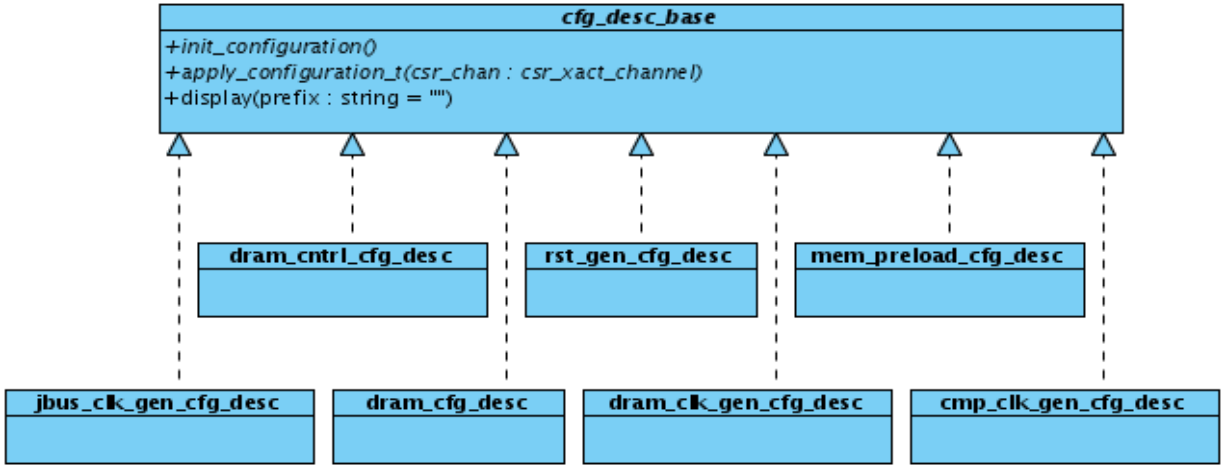
Configuration Descriptors

- The Composition Approach
  - To resolve this we can employ the following design principle "program to an interface not an implementation".



Configuration Descriptors

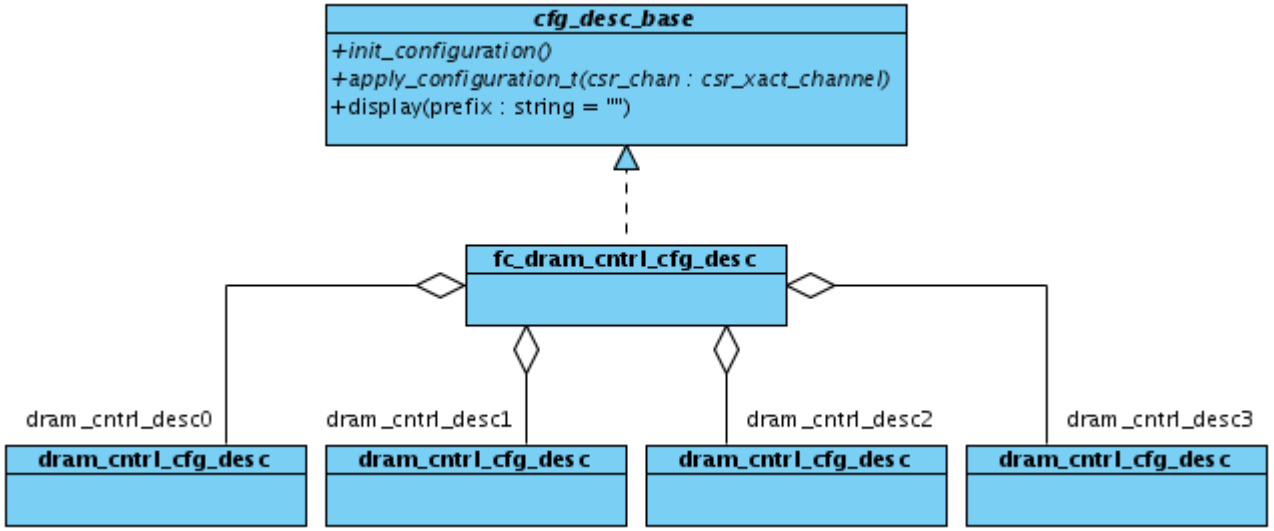
- The Composition Approach
  - To resolve this we can employ the following design principle "program to an interface not an implementation".



  - Anything that inherits from the pure virtual base class will need to implement all its methods.
  - All an environment needs to do is to call the init and apply methods appropriately.

Configuration Descriptors

- The Composition Approach
  - An FC cfg_desc can be composed of the SAT cfg_desc, and any extra ones it might need.



  - The FC environment doesn't need to know any information about how to configure the sub blocks, especially if parameter containers are used.

Configuration Descriptors

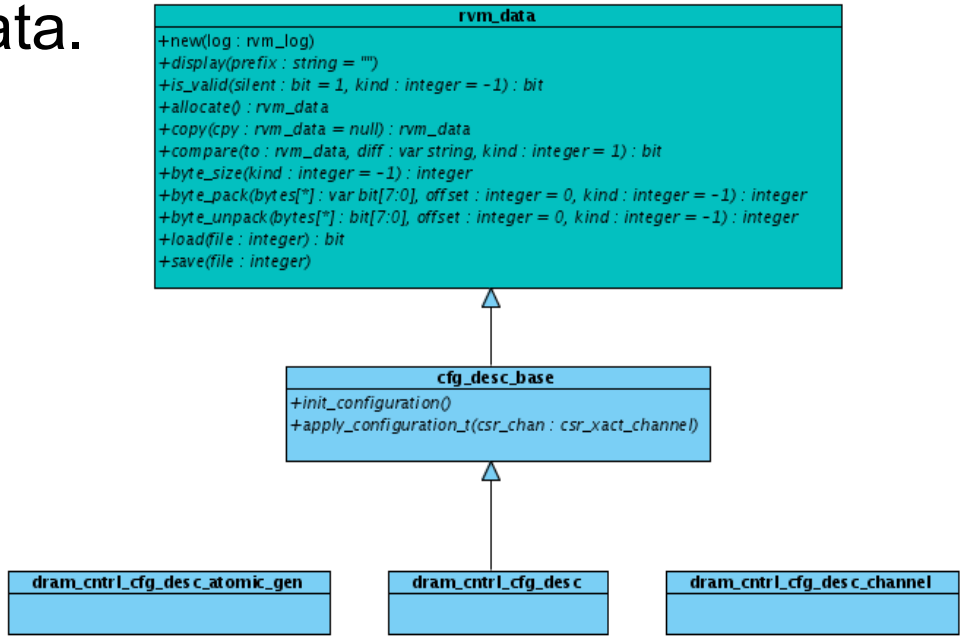
- The Composition Approach
  - The down side to this solution is that it is relatively static.
    - ➢The environment is explicitly tied to and manages the available descriptors.
    - ➢A test would need to derive a new descriptor and replace the environments copy with it, or control the descriptors programmatically.
    - ➢If a test wanted to change configurations mid-simulation it would need to perform all the actions of the rvm_env::cfg_dut_t method.

Configuration Descriptors

- The Factory Approach
  - RVM provides all the tools we need to generate our cfg_desc objects from blueprints.

Configuration Descriptors

- The Factory Approach
  - All that needs to be done is for the descriptors to be derived from rvm_data.



  - This will give us the following for free:
    - A macro to create an atomic generator for the descriptors
    - A macro to create a channel to communicate them.

Configuration Descriptors

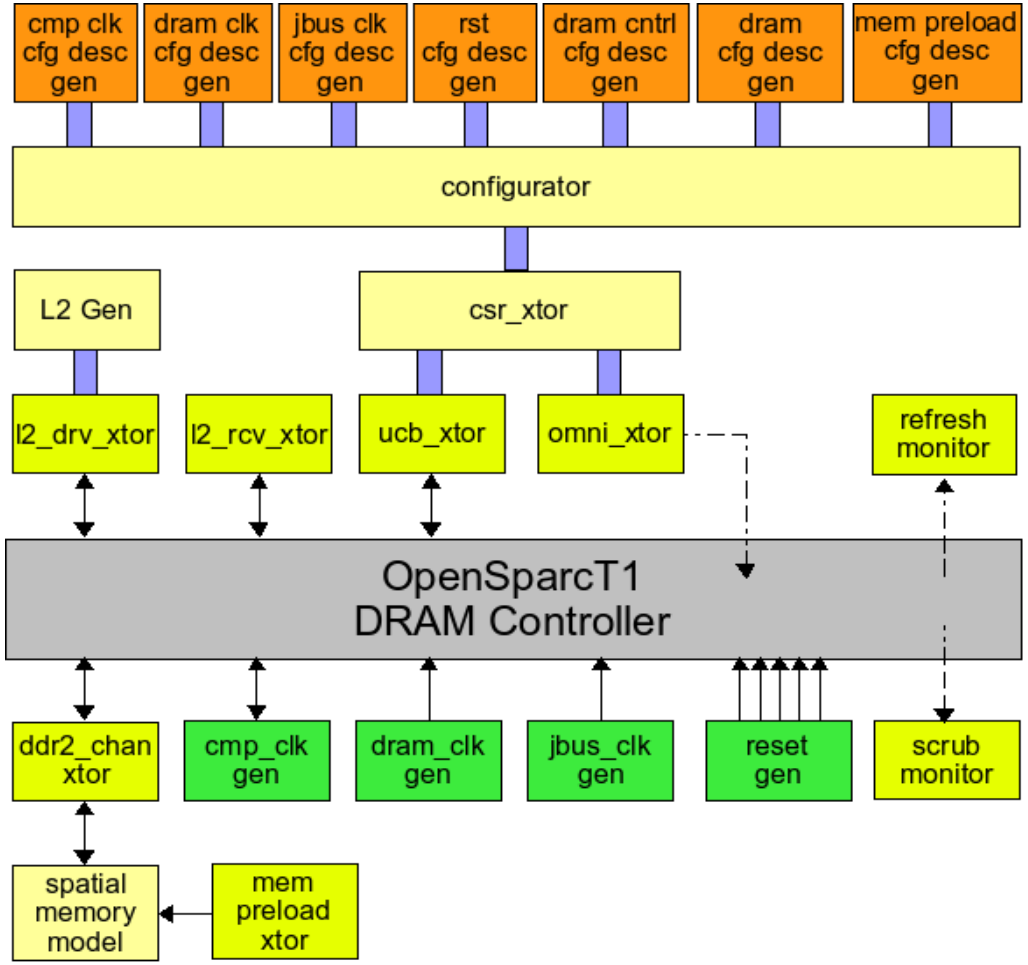
- The Factory Approach
  - Each of the cfg_desc instances in the environment can now be replaced with an atomic generator that creates them from a constrained random blueprint.
  - The only thing missing is something to consume the generated cfg_desc's and to apply them. This is where the “configurator” comes in.

Configuration Descriptors

- The Factory Approach
  - Each of the cfg_desc instances in the environment can now be replaced with an atomic generator that creates them from a constrained random blueprint.
  - The only thing missing is something to consume the generated cfg_desc's and to apply them. This is where the “configurator” comes in.
    - ➢The configurator is a transactor that operates on a given smartQ of configuration descriptor channels.
    - ➢It consumes one cfg_desc off each channel and invokes their apply_configuration_t methods.
    - ➢Once the smartQ has been iterated over the configurator stops itself.

Configuration Descriptors

- The Factory Approach



Configuration Descriptors

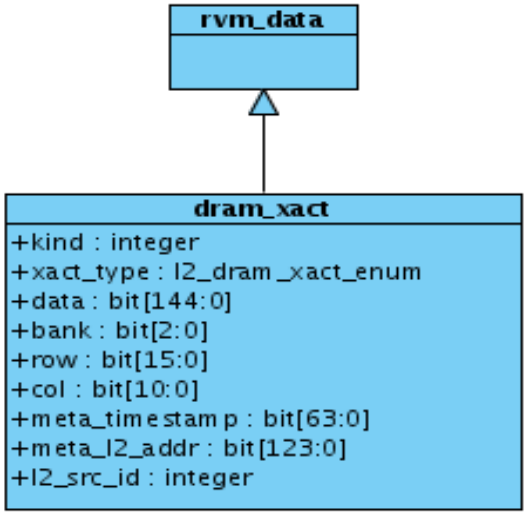
- The Factory Approach
  - This solution has many benefits:
    - A test can replace a generator's blueprint with its own version of the configuration.
    - A test can replace a generator's blueprints mid-simulation, flush the generator channels and start the configurator again.
    - A test can create a whole new smartQ of cfg_desc channels and pass it to the configurator.
    - A library of derived configuration descriptor classes can be developed and shared between environments.
  - It could also be used to manage common parameters.

Meta Data

- The Problem
  - Meta data is any information that is used by the testbench to annotate and organize the state of the simulation e.g.
    - Timestamps
    - Tracking Ids
    - Error Descriptors
  - Data is used by the verification environment to stimulate interfaces, and to check responses to that stimulus. Whereas meta data is used by the environment to:
    - Trace transaction details particular to the environment.
    - Communicate “out of band” information between components.
  - Different environments require different meta data, which can make reusing objects difficult.

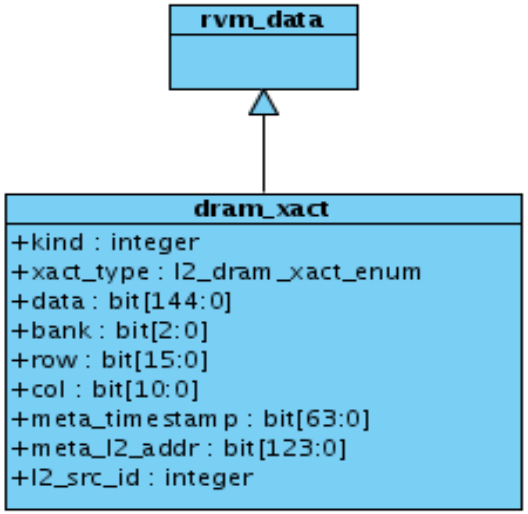
Meta Data

- The Coupling with Data Approach
  - Its tempting to embed meta data into the data classes themselves.



Meta Data

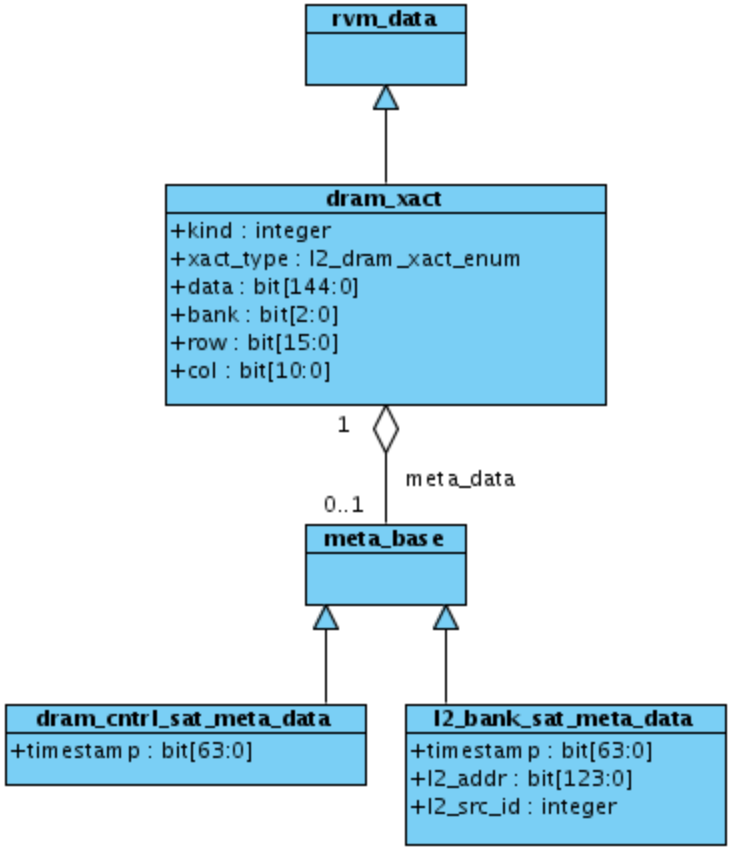
- The Coupling with Data Approach
  - Its tempting to embed meta data into the data classes themselves.



  - Reusing a SAT developed data object in the full chip environment can be difficult since the extra contextual information required is different.
  - The data object becomes a catch all for any extra information that needs to be tracked.

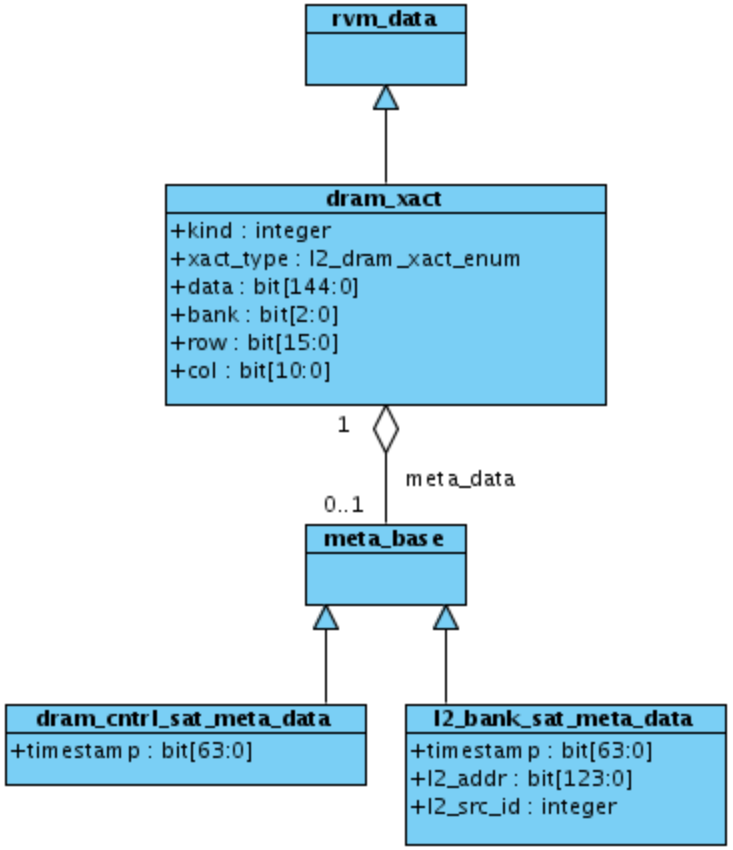
Meta Data

- The Dynamically Composing Functionality Approach
  - Create an empty 'place-holder' in all our data objects of type meta_base.



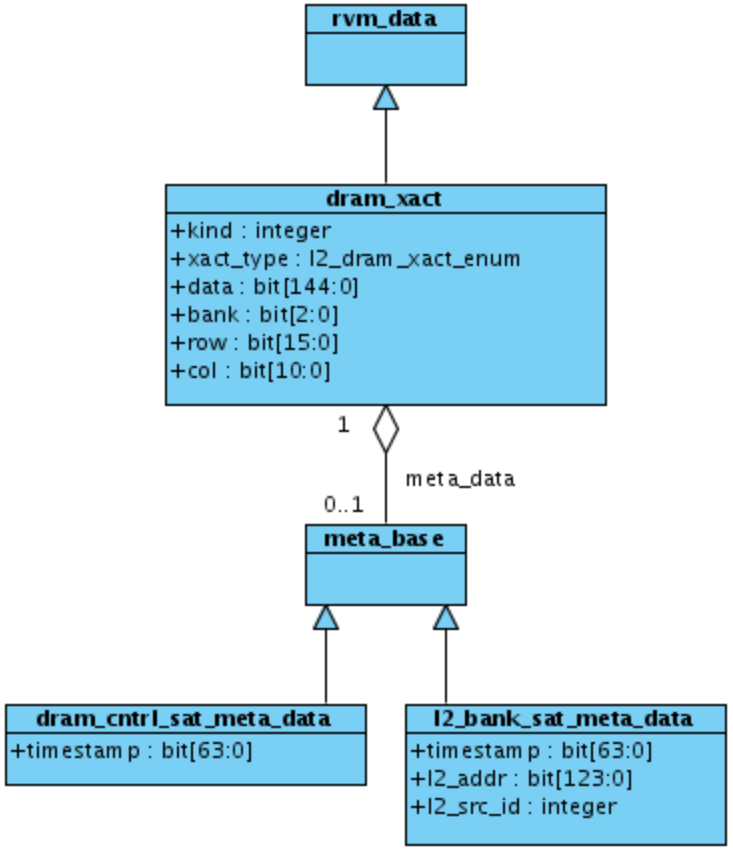
Meta Data

- The Dynamically Composing Functionality Approach
  - Create an empty 'place-holder' in all our data objects of type meta_base.
  - The meta data is now decoupled from the data objects.



Meta Data

- The Dynamically Composing Functionality Approach
  - Create an empty 'place-holder' in all our data objects of type meta_base.
  - The meta data is now decoupled from the data objects.
  - Environment components can create any number of derivative meta data objects and attach them to the data objects dynamically.



Conclusion

- Verification is getting harder. Utilizing tried and tested software design principles helps to reduce complexity and allows us to focus on debugging the design rather than our testbench code.
- The RVM framework provides a good starting point for pattern-based testbench design, however it doesn't solve everything. One can still architect oneself into a corner.
- The GoF OOP design patterns have been highly influential in the domain of software engineering. Sharing verification patterns as a community could arguably revolutionize our approach to verification.

Conclusion

- Developing your own arsenal of generic reusable verification patterns will pay dividends in the long run.
- Always consider decoupling and OCP when designing verification environments and their components.
- opensparc.net and opencores.org give us non-proprietary designs to demonstrate our ideas.

Conclusion

- There are a number ideas covered by the associated paper.
  - Checkers and Scoreboards
  - Error Injection
  - Abstract Functional Coverage
  - Environment Composition

Acknowledgements

- Various current and ex Sun Microsystems, Inc employees who have played a role in putting together these ideas, testing them or at the very least have sat through all the arguments.
- Janick and Synopsys' great work.

References

- OpenSPARC T1 Microarchitecture Specification, Sun Microsystems, Inc. Part No. 81906650-10 (www.opensparc.net).
- Head First Design Patterns, O'Reilly Media Inc. ISBN: 0-596-00712-4
- Design Pattens: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series. ISBN: 0-201-63361-2

Backup Slides

Error Injection

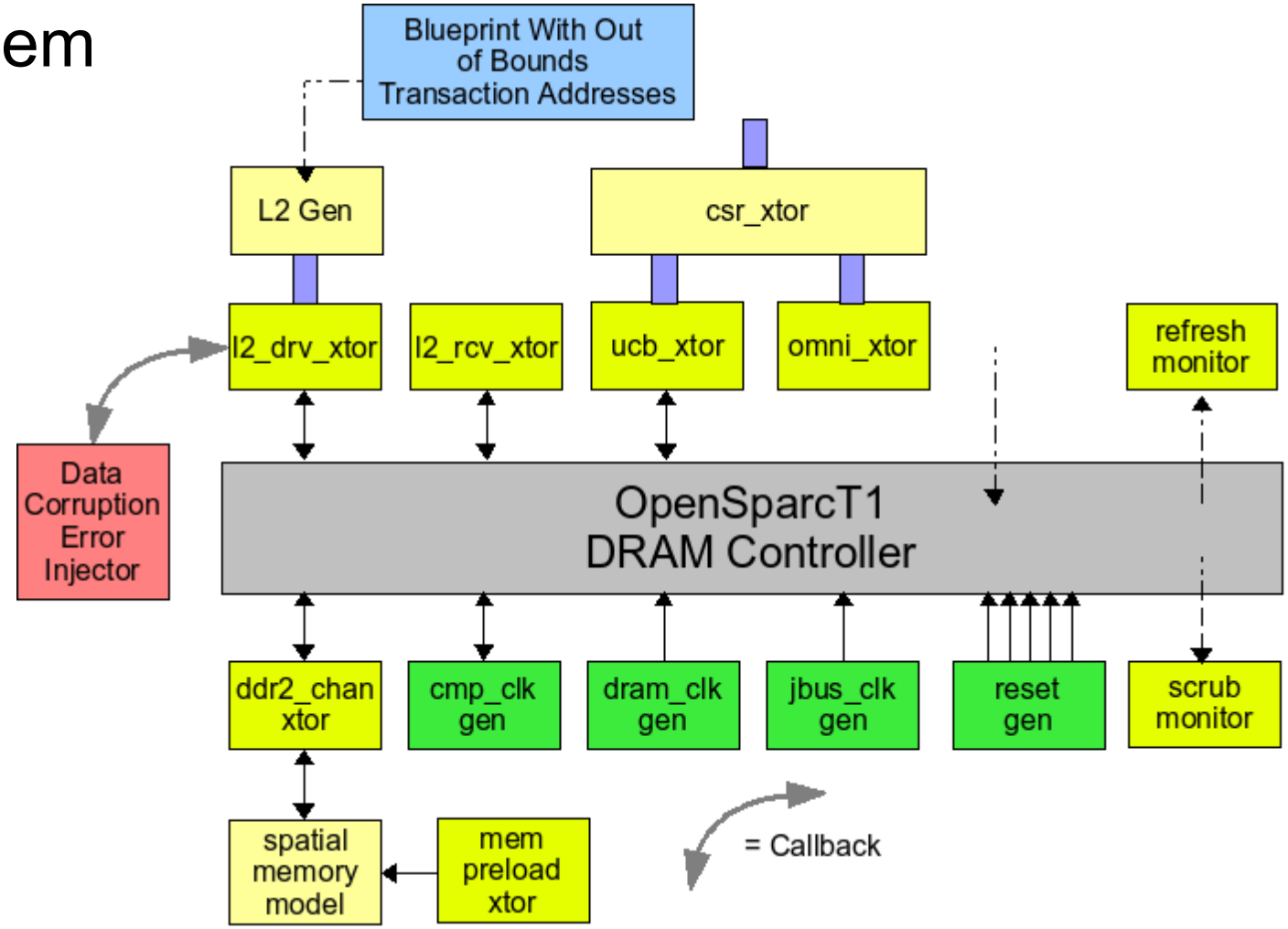
- The Problem
  - The RVM manual documents two types of error injection:
    - Illegal Transaction Generation

      This is achieved by constraining the generators to produce transactions that are erroneous with respect to specified legal packets.
    - Protocol and Interface Corruption

      This is achieved by attaching error injectors to transactors via callbacks. The error injectors decide whether or not to manipulate a packet itself or to instruct the transactor to corrupt the packet.

Error Injection

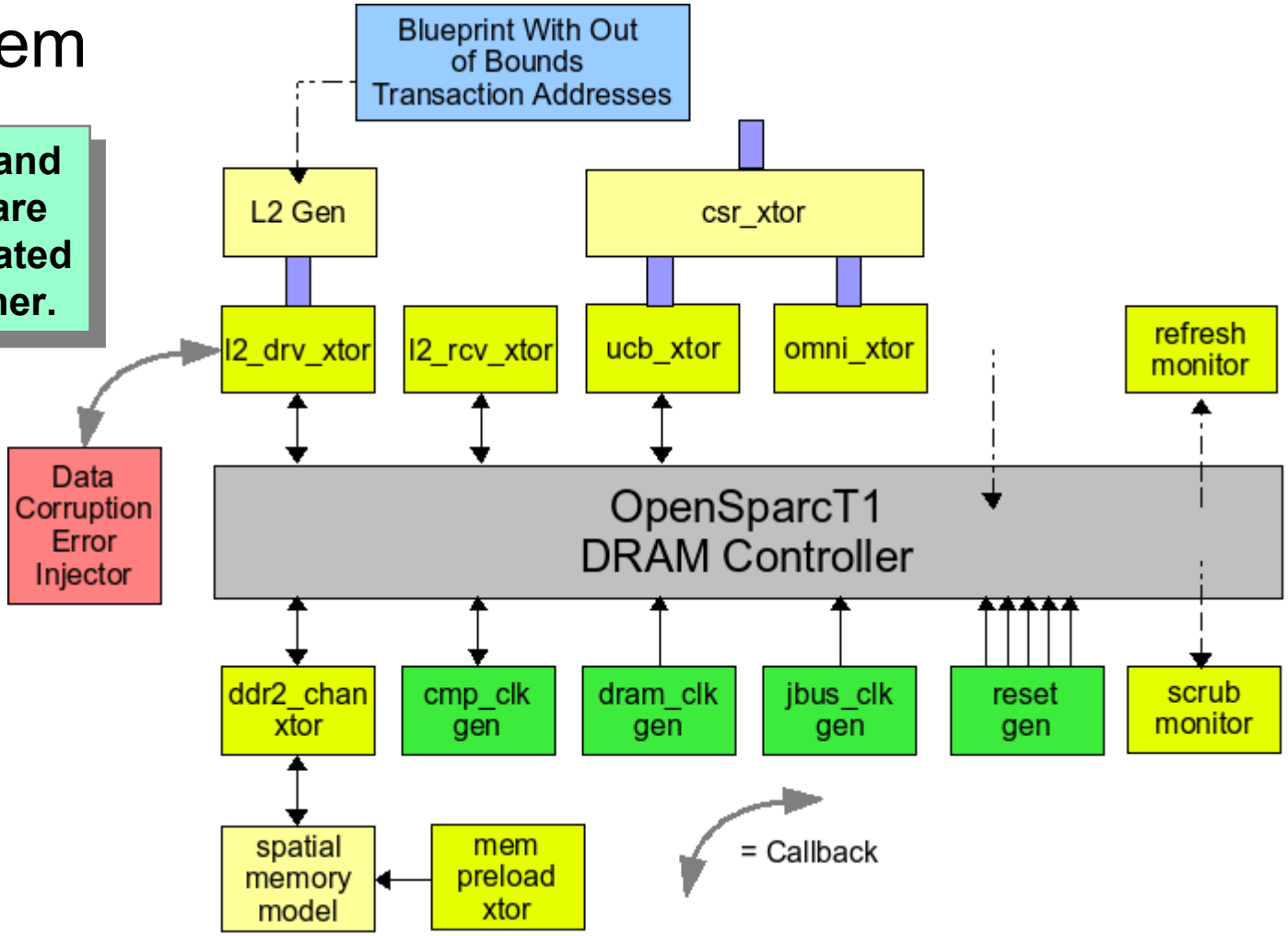
- The Problem



Error Injection

- The Problem

The generator and error injector are completely isolated from one another.



Error Injection

- The Pure Constrained Random Approach
  - Change error injector into a generator, thus allowing a test to control it via a blueprint (error descriptor) rather than it free wheel on its own.
  - Error injection generator is hooked up via callbacks to the transactors as normal.
  - Every time they receive a packet they randomize their error descriptor to determine how to corrupt the packet.
  - This solution provide tests with more control over the error injectors, however the packet generator and error injector blueprints need to be synchronized.

Error Injection

- The Generator Slave Approach
  - Provide error injector generator with error descriptor blueprint via meta data attached to the packet being corrupted.
  - Test can instruct transaction generator to create packets with respect to given blueprint, and to attach an appropriate error description blueprint to the transaction meta data place holder.
  - If the error injection generator receives a transaction carrying an error description blueprint it will use it to corrupt the packet, otherwise it uses it's default blueprint.
  - A test can now setup a default error injection scheme and associate errors with specific packets.