Verification of Transaction Level Models of Embedded Systems

Abstract

As complexity increases in embedded systems design, there is need for more time for verification purposes. For embedded systems, the only verification that can be done is running test cases, and the number of cases increases exponentially. In order to shorten this verification phase of the design, we propose a methodology to do formal verification of embedded systems. In formal verification no test cases are needed, but an mathematical analysis of the original model and the refined one. We base our tool on the Model Algebra theory of embedded systems, and apply its transformation rules to our models to check for equivalency. We test this transformation rules in various scenarios and prove that it is a promising methodology to improve embedded system design.

Keywords: Embedded Systems, Formal Verification, Transaction Level Models.

1 Introduction

1.1 Verification in the Traditional Design Flow

In the traditional hardware design flow, based on a specification (or "golden") model, a Register-Transfer-Level model is created manually. This phase introduces errors in the design, since it's done manually, so the developed RTL must be verified against the "golden" model. This step is called the verification process. This is illustrated in Figure 1.

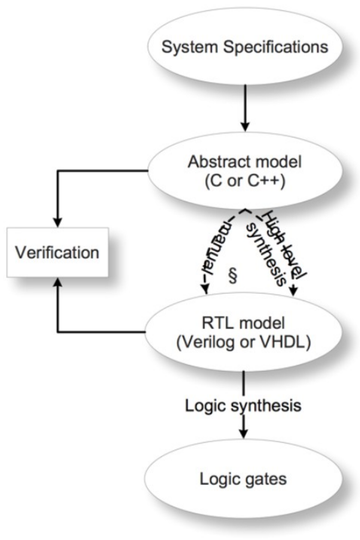


Figure 1: Verification phase in the design flow

1.1.1 Verification Challenges

The verification process faces the same challenges as the design process: due to the increasing complexity of embedded systems, both have to explore new approaches to deliver their tasks in shorter time-to-market time frames. Nowadays, verification time consumes more than half of the design process stage. There are several reasons causing this: for one part, higher complexity of designs mandate longer verification phases. Since most of the verification is based on simulations, the low speed of co-simulations slows down the process. Another consequence of the higher complexity is the number of test cases needed to functionally verify the model. This number of test cases increases dramatically, and its creation and selection is time consuming.

1.1.2 Verification Methods

We can classify verification methods into two: simulation-based methods and formal methods. Simulation based verification takes a set of inputs, feeds them into the model, and compares the set of outputs to the expected or "correct" set of outputs, usually obtained from the golden model. The model is referred as DUT or Device Under Test. This method is the most commonly used, as mentioned above, and it is an extremely time-consuming process. The set of test cases may be incredibly large or almost infinite, so it is up to the verification engineer to select and write the most “representative” cases to use. In summary, simulation based methods are lengthy and very labor-intensive. The second verification method is formal verification. Here, simulation is not needed. Mathematical methods are used to verify a model. There are three approaches: equivalence checking, model checking and theorem proving. Equivalence checking tests for correctness of synthesis and optimization of models, model checking tests a formal representation of a model to see if a specified property is satisfied and finally, theorem proving takes the model’s mathematical representation of a specification model and an implementation model and proves equivalence by using axioms and inference rules.

1.2 System Level Design and TLMs

Due to the challenges mentioned above, the level of abstraction of the designs has moved up to the System Level. Raising the level of abstraction from RTL to System Level greatly reduces the number of components, thus speeding the design time, simulation time and design space exploration time. Recently, TLMs have become the optimum way of doing system level design.

For TLMs to be synthesizable and verifiable, well defined TLM semantics are required. Existing formalism for RTL design such as FSDM or boolean algebra are insufficient to express TLMs. Therefore, new formalisms for TLM based design are needed. Furthermore, the TLM semantics must allow simulation models written in languages like SystemC to be easily abstracted into mathematical expressions for symbolic manipulation.

Model Algebra [1,2,3] is one such formalism that can be used for refinement and verification of Transaction Level Models. The basic concept is the separation of functional and architectural modeling. Executable models are defined in the functional space which platform netlists are defined in the architecture space. Subsequently, a mapping is defined from the functional to architecture space, that allows designers to evaluate useful metrics. A key assumption is a many to one mapping from the function space to the architecture space. This constraint is necessary to produce an unambiguous design.

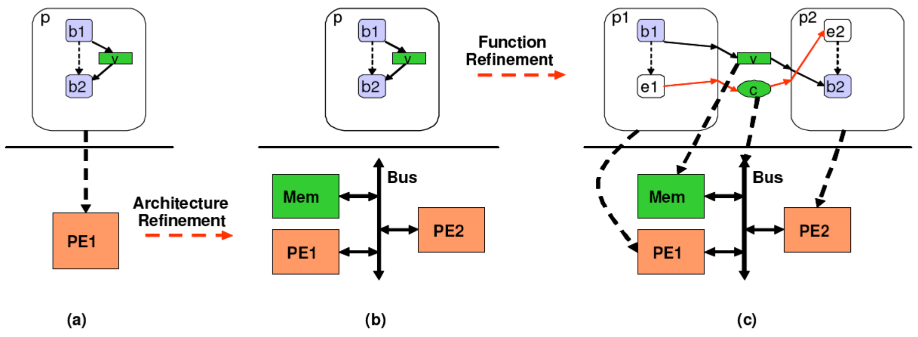


Figure 2: Architecture and functional refinement in platform based design

Figure 2(a) shows a simple mapping of a sequential composition of two functional objects (called behaviors) onto a processing element (PE) in the architecture. Now, assume that the designers figures that this mapping does not produce a satisfactory execution time. So, he or she may select another PE (PE2) that is optimized for behavior b2. The new architecture is shown in 2(b). However, in this new function and architecture specification, there is no feasible mapping. This is because a sequential composition cannot be mapped to a concurrent architecture. Therefore, the function must now be refined to the one shown in 2(c) by isolating b2 into a concurrent process. A synchronization channel is added to keep the execution order between b1 and b2. This new refined functional model is now mappable to the refined architecture.

As we saw in the optimization example above, functional models may need to be refined every time the architecture netlist is modified. It is imperative that each such refinement be functionality-preserving. This poses the TLM functional equivalence problem as illustrated in Figure 3. The problem is to verify that any functional refinement produces a TLM that is functionally equivalent to the original TLM. In this article we propose such a tool based on Model Algebra that verifies equivalence of the two well-formed TLMs. By well-formed, we mean that the TLMs must follow the semantics of the objects and composition rules of Model Algebra. Here, we present new transformation rules that will allow more refinements in the models. In addition, we developed specific application programming interfaces (APIs) that facilitate construction of algebraic TLMs and to perform symbolic transformations on them in conformance with the rules of Model Algebra.

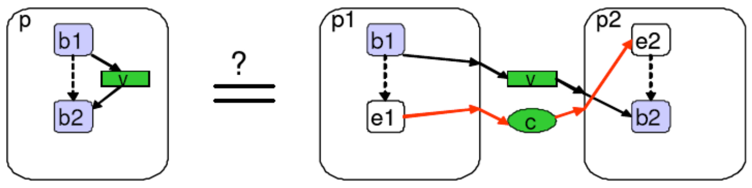


Figure 3: TLM equivalence verification problem

2 Model Algebra

Model Algebra is a formalism for refinement and verification of TLMs. A formalism is a set of objects and composition rules that represent the relationship between these objects. Model Algebra aims to express executable system models with structural details, using the same composition rules and objects. Model Algebra's theory was published in [1], so only a brief summary will be presented below.

2.1 Objects

The objects of Model Algebra consist of: Behaviors, Channels, Variables, Behavior interfaces, and Behavior ports. Model Algebra can be represented graphically in a *Behavior Control Graph (BCG)*,[1] which shows the control flow of any model in MA. In BCG, there are two types of nodes: behavior nodes and control flow nodes.

Behaviors: They are the computational units of Model Algebra. It has *ports* that allows it to connect to other behaviors using *interfaces*. It is drawn using a rounded rectangle, shown in Figure 4. They can be hierarchical, meaning that they can contain other behaviors and any other MA objects inside. There is also a special kind of behaviors called *Identity* behaviors, which read data from its *in* port and writes it out to its *out* port. It essentially does not do any kind of computation inside it, it only forwards the data from one port to another. Two notable identity behaviors present in any hierarchical behavior are the Virtual Starting Point (VSP) behavior and the Virtual Terminating Point (VTP) behavior. As their names specify, computation starts at the VSP and ends at the VTP.

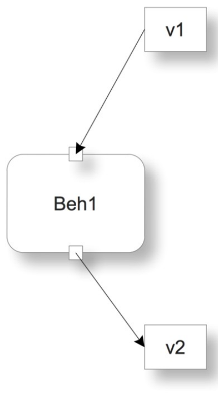


Figure 4: Behavior in Model Algebra

Channels: Are the communication elements that go from behavior to behavior.

Variables: Allow communication by the way of storing information and being read by a behavior. In its graphical representation, it is draw as a rectangle. In Figure 4, the behavior *Beh 1* reads from a variable *v1* and writes it to the variable *v2*.

2.2 Composition Rules

Composition rules are the relations between objects in Model Algebra.

Control flow

It determines the execution order of behaviors during a simulation. All predecessing behaviors must finish and a condition be true in order for the successor behavior to start. It is formally expressed as:

$$q : b_1 \& b_2 \& \ldots \& b_n \rightarrow b \qquad (1)$$

This expression states that for behavior *b* to execute, the condition *q* must be *true* and the behaviors must all execute first. It is represented graphically as a circle with edges pointing towards it from the preceding behaviors and has an edge toward the successor behavior. This can be seen in Figure 5.

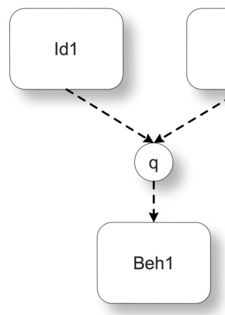


Figure 5: Control Node

Blocking and non-blocking operations

Behaviors may use their ports to read or write to/from a variable or another port. These read and write operations may be either blocking or non-blocking operations. We will refer to these operations as *data dependencies*. They are represented as straight arrows between ports and variables.

2.3 Transformation Rules

A model can be transformed by rearranging and replacing objects. Model refinement is basically a well-defined sequence of transformations. If each of these transformations is proven correct in a formal context, then we can perform verification by correct refinement with two given models. Starting with a test model *M*, a refined (stated as the function *R()*) model *M'* can be defined as the successive transformations on *M*:

$$M' = R(M) = t_n(t_{n-1}(..t_1(M)...) \qquad (2)$$

2.3.1 Hierarchical Behavior Flattening

Since we only need the leaf level behaviors, hierarchical behaviors can be “flattened” to reveal its sub-behaviors. Basically there are two types of objects that are modified in the flattening process: control dependencies and data dependencies. Control dependencies are modified because once the hierarchical behavior has been flattened, the control dependency leading to it is directed instead to its VSP behavior, and the VTP will take the place of the as the predecessor in all Control Dependencies that included the parent behavior. Data dependencies are modified when all ports of the parent behavior are flattened out. The data dependency is merged with the port data dependency inside the parent. This includes the links in all channels, extending the channel to the subbehavior.

2.3.2 Identity Elimination

Since the identity behavior does not perform any computation, it can be removed from the model after resolving data and control dependencies. Any variables being read and written by the identity behavior are merged, and the ports removed. The control nodes preceding the behavior are merged with the control nodes succeeding the identity. This is illustrated in Figure 6. Unresolved channel dependencies in an identity behavior prevents it from being eliminated; channel resolution has to be applied first before removing its ports.

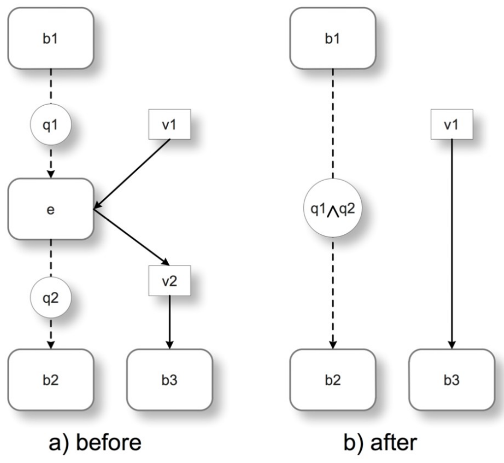


Figure 6: Identity Elimination Rule

2.3.3 Redundant Control Dependency Elimination

This rule removes unneeded control dependencies. Key to this rule is the concept of *dominance* of behaviors: if a behavior *A* always executes at least once for every execution of behavior *B*, then we can se that *A* is a *dominator* of *B*. Now, given 3 behaviors *A*,*B* and *C*, shown in Figure 7(a), since *A* dominates *B*, we know that for every execution of *C*, the other two behaviors have already executed, hence, the control edge from *A* to *C* is unnecessary, so it can be eliminated. This can be seen in Figure 7(b).

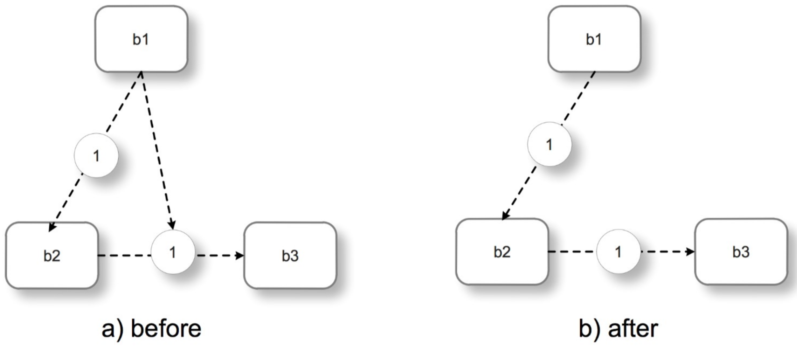


Figure 7: Redundant Control Dependency Elimination Rule

2.3.4 Control Relaxation

Given a model in Figure 8(a), if there is no data dependency between behavior *b1* and behavior *b2*, and the control node between them has no other port or variable dependency, then *b1* and *b2* can execute concurrently since the order of their executions does not alter any variable trace. Both behaviors would still continue on to behavior *b3*, as seen in Figure 8(b).

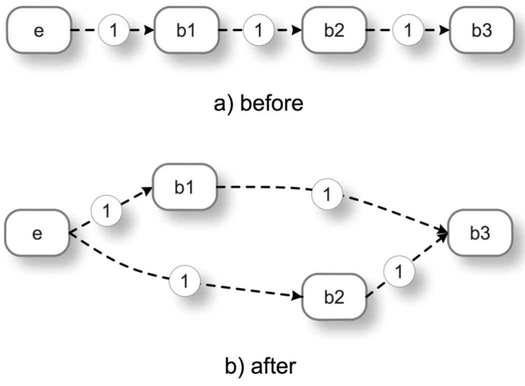


Figure 8: Control Relaxation Rule

3 Transaction Level Modeling

In order to accurately model TLMs, we improved some aspects of Model Algebra and included new transformation rules.

3.1 Channel Resolution Rule

We refined channels to be point to point between two behaviors, and will have double handshake semantics. These behaviors must be either hierarchical behaviors or identity behaviors. The channel will also no longer hold addresses, and any data to be transferred will be read by the identity behavior on one side and written out by the identity behavior on the other side. It is best illustrated in Figure 9.

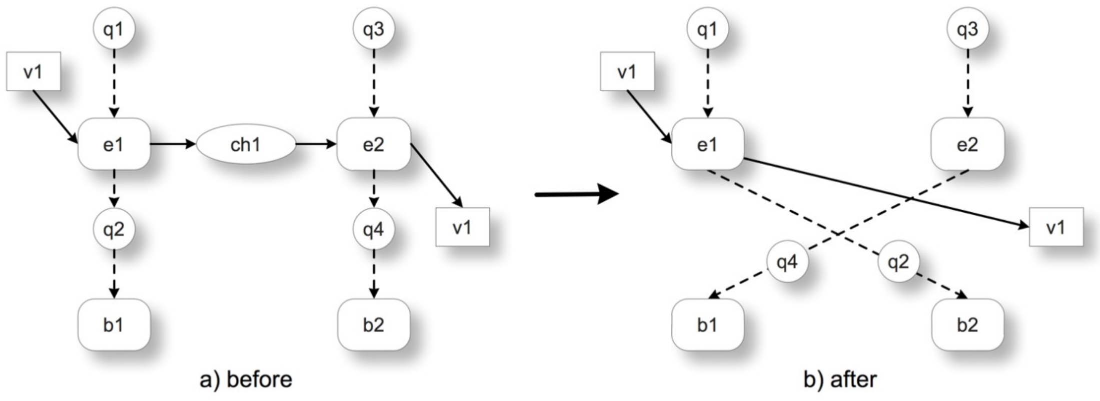


Figure 9: Resolution of channels into control dependencies

3.2 Principle of Duplication

When designing embedded systems, the designers may encounter modules that slow down the execution flow because of their slow hardware response. In Figure 10 a), let’s say that module B is the one with a big delay. One of the common solutions would be to simply duplicate the module in question (hardware IP for instance) and do a parallel execution to speed the execution, as shown in b). Module A will forward half of its computation to module B’ and half to B”, assuming that both modules perform identical computation tasks. Finally, our system is shown in c), with 2 modules B’ and B” in an endless loop, performing calculations for A and forwarding the result to C.

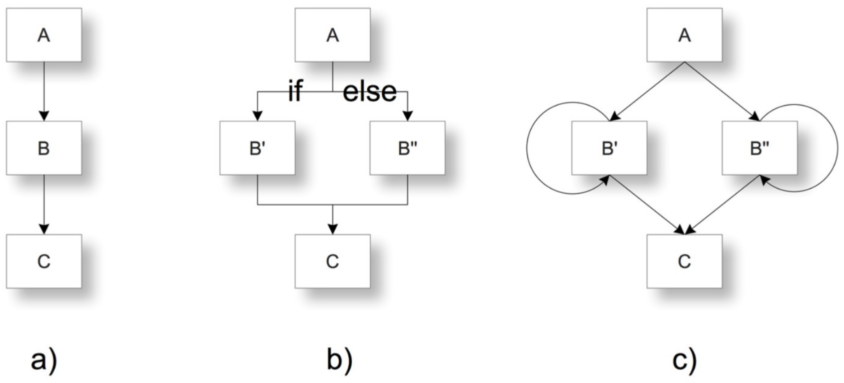


Figure 10: Principle of duplication

3.3 Behavior Type

The principle of duplication shown in the previous section is sound if and only if the two behaviors running in parallel are identical. In order to include this property into Model Algebra, we introduce *Behavior Type*: each behavior will have a type, and may be shown explicitly after the behavior name, separated by a colon. Figure 11 updates the figures in c) (from Figure 10) with the behavior type and with the proper Model Algebra objects.

For two behaviors to be of the same type, there is one underlying condition: both behaviors have to have the same ports and these ports have to have the same bindings to other ports or variables. If this is not met, the two behaviors shall be considered of different type.

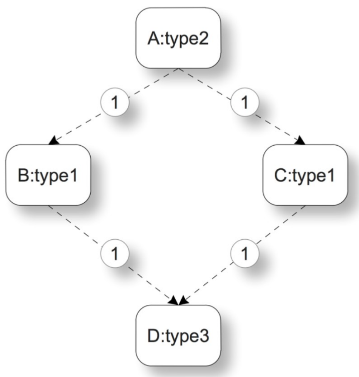


Figure 11: Behavior type

3.4 Behavior Merging

This transformation merges two behaviors if the following conditions are met:

1. Both behaviors are of the same type. This implies that both behaviors have the same bindings to the same ports and/or variables.
2. Their sets of successor behaviors are equal. In other words, the execution of any of the two behaviors is followed immediately by the execution of one specific behavior.

Given a Model M, let there be n control nodes (q_1 to q_n) in M, which have n edges to a behavior b_n.

Now, $\forall i$, *s.t.*$1 \leq i \leq n$.

In M, q_i has in-degree *I(i)*.

Let, $(b_1^i, q_i), (b_2^i, q_i), ..., (b_{I(i)}^i, q_i) \in E(BCG(M))$.

And b_i ... $b_{I(i)}$ are of the same type, therefore, $b' \in BCG(M)$ and $q_i \bigvee q_n : b' \rightarrow b_n \in BCG(M')$. This transformation rule is illustrated in 12.

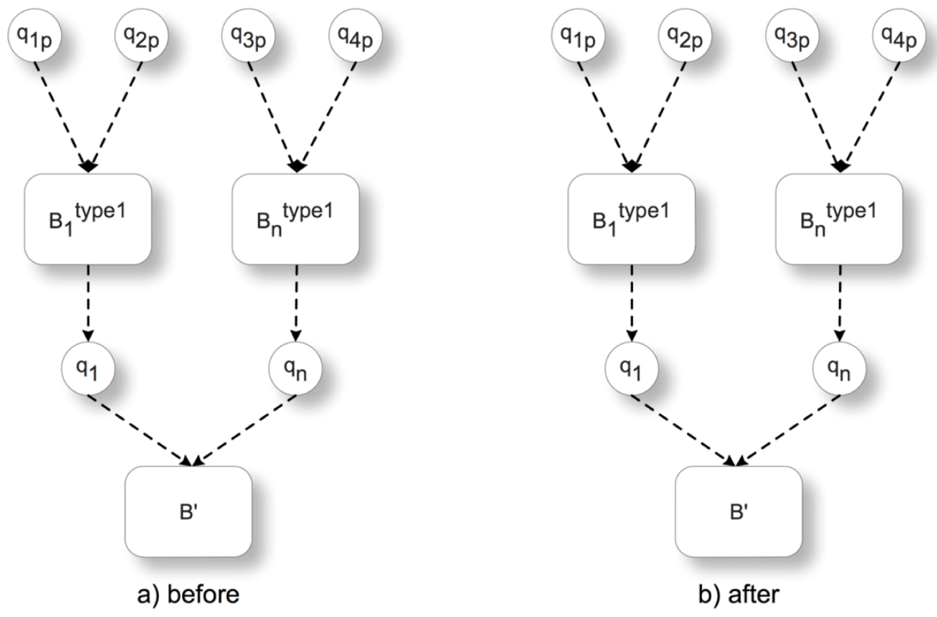


Figure 12: Behavior Merging rule

3.5 Control Node Merging

This transformation rule allows to reduce the number of control nodes by merging two or more nodes if:

1. Their sets of predecessor behaviors are equal.

2. They have the same successor behavior.

Given a Model M, let there be n control nodes (q$_1$ to q$_n$) in M, which have n edges to a behavior b'.
Let there be a set of m behavior nodes (b$_1$ to b$_m$) in M.
Now, $\forall i, s.t. 1 \leq i \leq m, \forall j, s.t. 1 \leq j \leq n$.
Let, $(b_1, q_1), ..., (b_m, q_1), ..., (b_1, q_n), ..., (b_m, q_n) \in E(BCG(M))$.
Therefore, $q_1 \bigvee ... q_n : b_1 \& ... \& b_m \rightarrow b' \in BCG(M')$.
This is illustrated in 13.

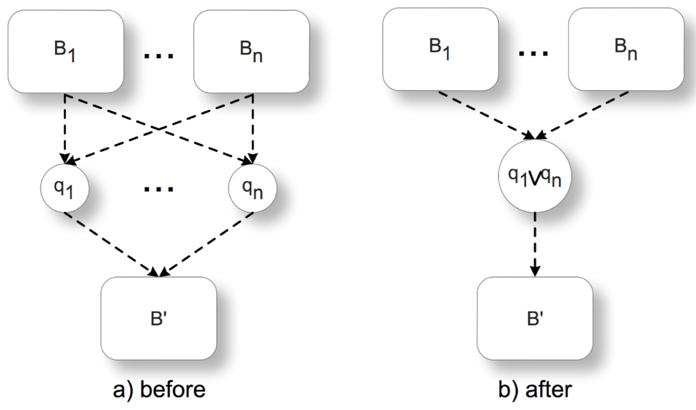


Figure 13: Control Merging rule

4 Model Algebra Data Structure

In order to be able to properly manipulate and transform a model described in Model Algebra, we need a suitable data structure to describe any posible model. We now describe the Model Algebra Data Structure used in our tool. Our Model Algebra Data Structure is saved in the XML format [4] and have the extension .MAG. The set of rules that all MAG files conform to is expressed in a XML Schema Definition (XSD) [5]. MAG files are composed of the following elements: *BEHAVIOR*, *VARIABLE*, *CONDITION*, *CHANNEL*, *LINK*, *PORT*, *CD*, *DD_VAR_NB_READ*, *DD_VAR_NB_WRITE*, *DD_PORT_NB_READ*, *DD_PORT_NB_WRITE*, *DD_PORT_B_READ*, and *DD_PORT_B_WRITE.*

The root of every MAG file is a *BEHAVIOR* object which contains any other objects listed above. The BEHAVIOR object is hierarchical, meaning that any object can only be contained inside a BEHAVIOR object and no other.

The graphical representation of the MAG object tree is shown in Figure 14. In this figure, we can see that the only other object that can be hierarchical is CHANNEL, which contains LINK. It describes the ports that connect both behaviors to the channel. CD refers to *Control Dependency*, and has an attribute pointing to CONDITION. This object marks if the CD depends on a variable, port or a boolean value. All the *data dependency* objects are represented in the figure as DD: these include data dependencies on variables and ports, blocking and nonblocking reads and write operations.

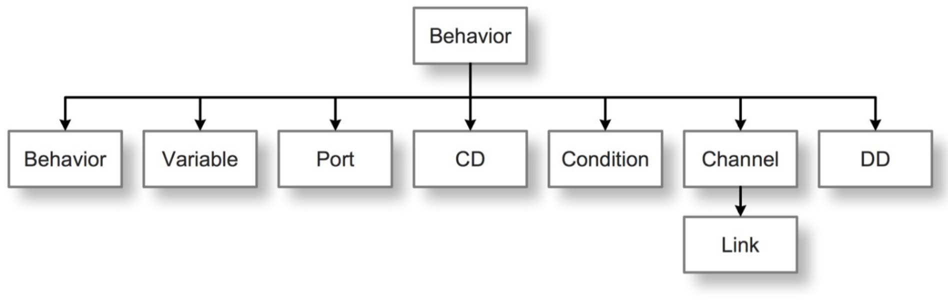


Figure 14: MAG Object tree

5 TLM Verifier

We developed a tool to create, transform and verify TLMs, named TLMVer. It is composed of:

1. Input API: it is the interface for the MA converter which creates a Model Algebra data structure. It creates all the MA objects and dependencies between them.
2. Frontend GUI: shows the graphical representation of the MAG file. It has an interface to allow the user to apply any transformation in any order to the model. It also checks for isomorphism between two models.
3. Backend: it responds to the GUI and applies all the transformation rules, and performs the equivalency checks.

In Figure 15, we show the verification flow using TLMVer. We would start with an application and goes through several refinement steps. Both applications' TLMs are fed into the MA Converter, which interfaces with our TLMVer API. This creates the MA representation (MAG file) that is the input for the TLMVer. Finally, the transformations are performed in this step by the tool and a isomorphism is checked at the end. If both models are isomorphic, we can say that both models are equivalent.

The MA Converter's task is to parse the TLM and make the appropiate calls to the Input API. This module has not yet been developed, and the models we use are created by a script calling the API.

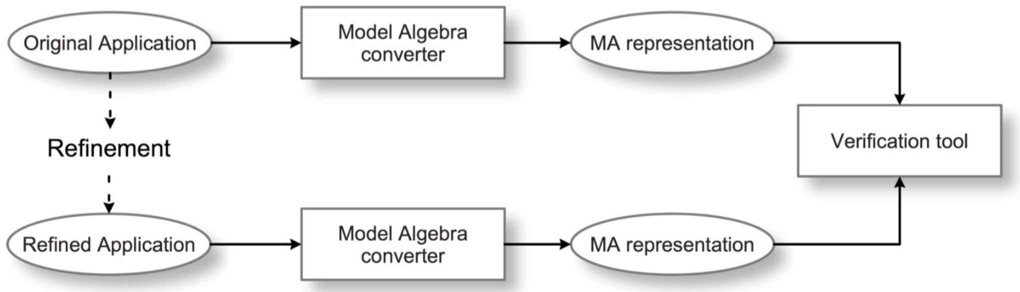


Figure 15: TLM Verifier tool flow

5.1 Isomorphism Checker

Our tool can take two models' MA representation and check if both graphs are *isomorphic*. Isomorphism indicates syntactic equality of 2 models and it is the strongest possible equivalence. MA representations have a *root* node which will be the Virtual Starting Point (VSP). They may or may not have a connected Virtual Terminating Point (VTP), and commonly has cyclic edges.

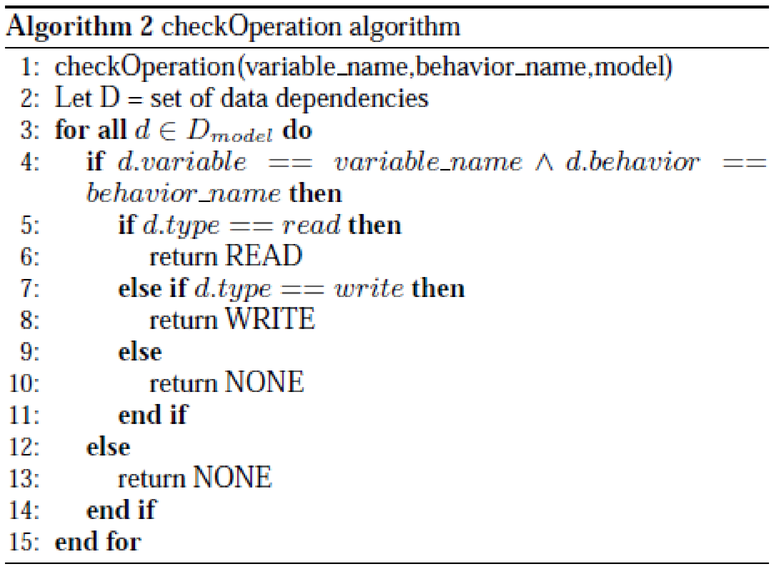
One assumption that makes the checker very simple is the case in which a model *M1* and a model *M2*, both share the same set of subbehaviors with the *same* name. In this case, the checker algorithm is shown in Algorithm 10. The checker's task is to mainly verify the data dependencies between each behavior and the variables, and to verify the *dominance* of each behavior to all other behaviors (line 10). If all dominance checks are true, and the data dependencies are equal, both models are *isomorphic*.

Algorithm 1 Isomorphism checker algorithm

```
1:  //Check the number of objects in each model
2:  checkNBehaviors(M1,M2)
3:  checkNControlDependencies(M1,M2)
4:  checkNDataDependencies(M1,M2)
5:  checkNVariables(M1,M2)
6:  Let B = set of behaviors
7:  for all $b_1, b_2 \in B_{M1}$ do
8:      $b_3$=findBehavior($b_1.name$, $B_{M2}$)
9:      $b_4$=findBehavior($b_2.name$, $B_{M2}$)
10:     if checkDominance($b_1, b_2$)!=checkDominance($b_3, b_4$) then
11:         return FALSE
12:     end if
13: end for
14: //check all data dependencies
15: Let V = set of variables
16: Let D = set of data dependencies
17: for all $d \in D_{M1}$ do
18:     $v_1 = d.variable$
19:     $b_1 = d.behavior$
20:     $v_2 = findVariable(d.variable, B_{M2}$
21:     $b_2 = findBehavior(d.behavior, B_{M2}$
22:     if checkOperation($v_1, b_1, M1$)!=checkOperation($v_2, b_2, M2$
        then
23:         return FALSE
24:     end if
25: end for
```

For each *variable/behavior* pair, a check operation is done to see if there exists a data dependency in which the behavior reads or writes to that variable (line 22). The function *checkOperation* is shown in

Algorithm 11.



Algorithm 2 checkOperation algorithm

```
checkOperation(variable_name,behavior_name,model)
Let D = set of data dependencies
for all $d \in D_{model}$ do
    if $d.variable == variable\_name \wedge d.behavior == behavior\_name$ then
        if $d.type == read$ then
            return READ
        else if $d.type == write$ then
            return WRITE
        else
            return NONE
        end if
    else
        return NONE
    end if
end for
```

6 System Level Refinement and Verification

The design methodology for our system is shown in Figure 16. It starts with an executable functional specification model of the design and is gradually refined into a cycle accurate model which is then forwarded into the traditional manufacturing phase. The refinement process is composed of several steps in which objects in our models are modified, replaced or eliminated. After each step, the designer ends up with a new executable model which serves as the base for the next refinement step. As shown in Figure 16, cycle accurate design is not part of our domain and will not be discussed here.

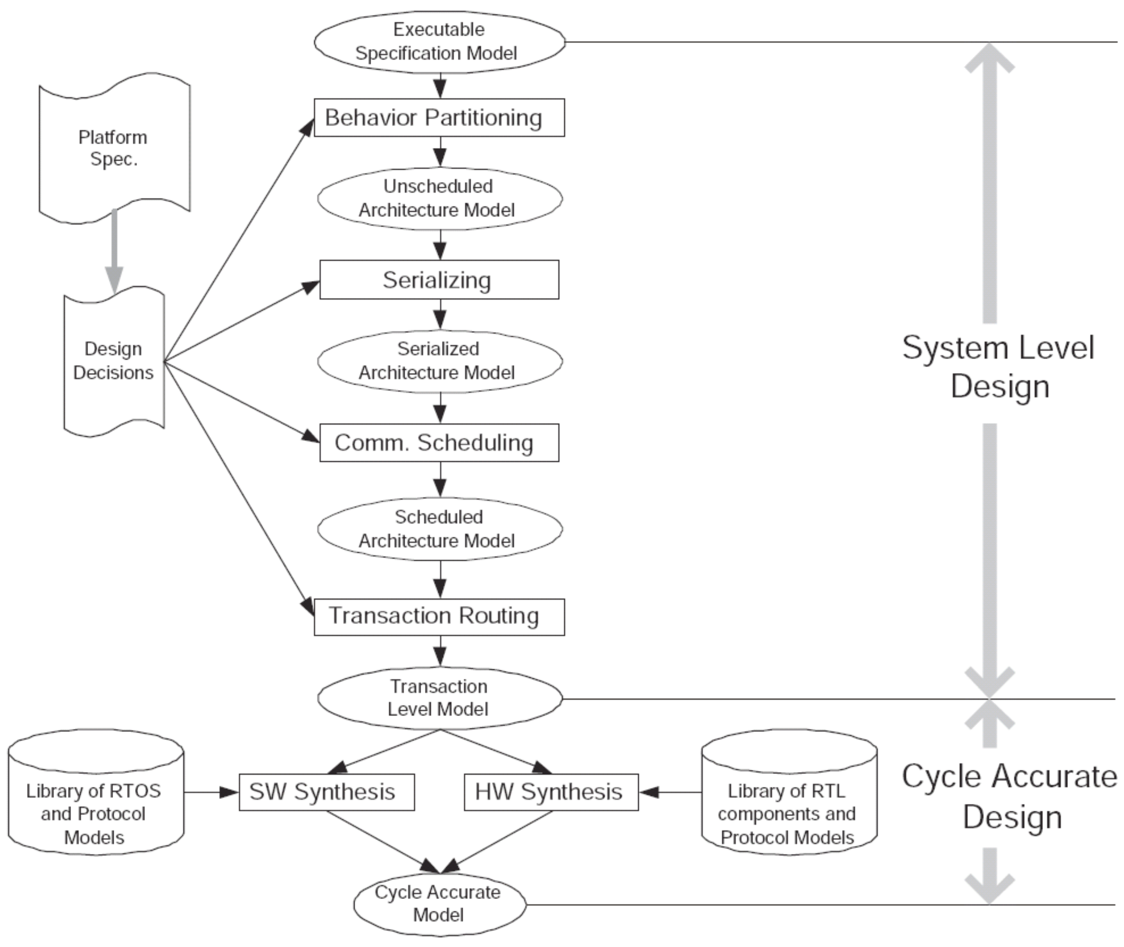


Figure 16: Refinement based methodology (courtesy of [5])

The refinement depicted in Figure 16 are:

1. Behavior Partitioning: the behaviors are rearranged to reflect the mapping of leaf behaviors to component behaviors.
2. Serializing: Behaviors that must be executed with a single controller are serialized, it converts parallel composition into sequential compositions.
3. Communication Scheduling: by modifying the scheduling of bus transactions, the performance of the design can be improved.
4. Transaction Routing: Splits transaction links into two links putting a router in between.

These key refinements were described in detail in [5]. In this article, we will focus on the design optimizations depicted in Figure 15. These optimizations can be proven for correctness using the new transformation rules of Model Algebra.

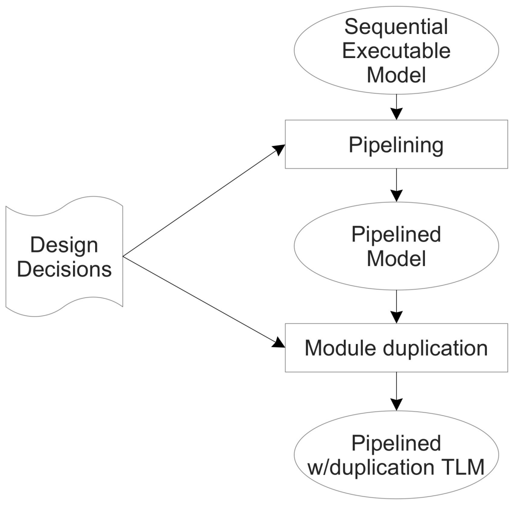


Figure 17: Optimizations

The optimization depicted are:

1. Pipelining: sequential behaviors are separated into different processing units which run concurrently. Each behavior will forward data to the next one and immediately begin processing the next data packet. Between each pair of behaviors a FIFO structure may be modeled.
2. Duplication: slow behaviors may be duplicated in order to compute two or more data packets simultaneously and speed up the pipeline.

6.1 FIFO modeling

To model communication between behaviors, we can use *communication channels*, described above. But for several types of applications, the model of computation used is Kahn Process Networks, were the behaviors execute concurrently and communicate through unbounded FIFOs. Our model of a FIFO can have one or more storage variables. Shown in Figure 18 is a 1-place FIFO. The structure is based on two identity behaviors: *e1* and *e2*. The first one performs the double handshake channel communication with the preceding behavior and writes out the data into the variable. The second identity behavior reads the variable and pass it through the channel to the next behavior.

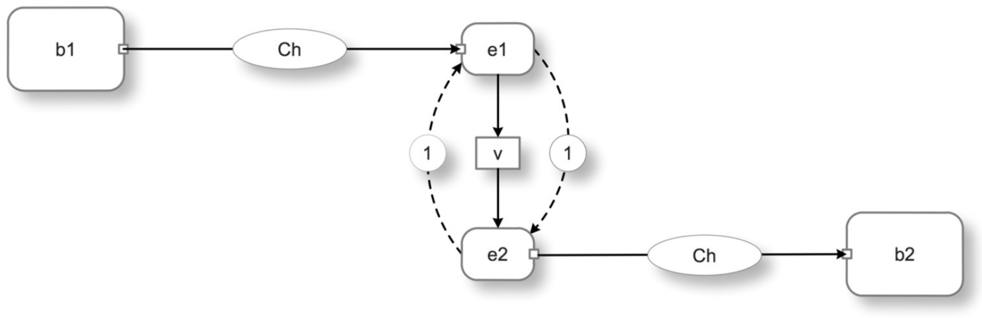


Figure 18: 1-place FIFO

A FIFO with more storage is constructed simply by repeating this structure. A n-place FIFO representation is shown in Figure 19.

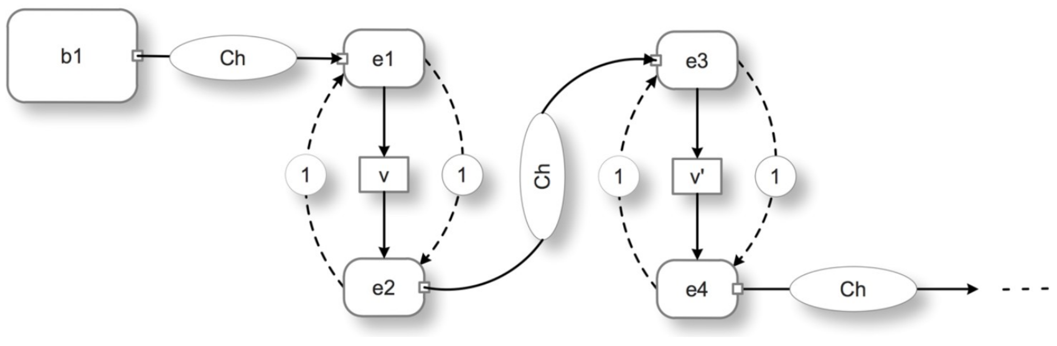


Figure 19: n-place FIFO

FIFO Transformation

We can use the transformation rules described in this article to prove that a n-place FIFO can be reduced to a 1-place FIFO. This is illustrated in Figure 20. The initial model is a subsection of a n-place FIFO depicting a 2-place FIFO, shown in the upper left corner of the figure. There is a incoming channel *ch1* and an outgoing channel *ch3*. The outgoing channel links with the rest of the FIFO. The first transformation rule applied is the Channel Resolution Rule, so channel *ch2* is resolved into two new control dependencies: identity behavior *e3* goes to *e1* and *e2* goes to *e4*. The Identity behavior *e2* now writes directly to variable *v2*. The result is shown in the upper right model in Figure 20. The next rule applied is Identity Elimination: *e2* is eliminated and the variables *v1* and *v2* are merged, keeping the name *v1*. The other rule that was applied is the Redundant Control Dependency Elimination: the control dependency between *vsp* and *e1* is eliminated. The result is shown in the lower right model in Figure 20. The last step is to apply again the Identity Elimination Rule to behavior *e3*. The resulting model is shown in the lower left part of the figure and it is the same as a 1-place FIFO, proving that any FIFO can be reduced into a 1-place FIFO applying the basic transformation rules of MA.

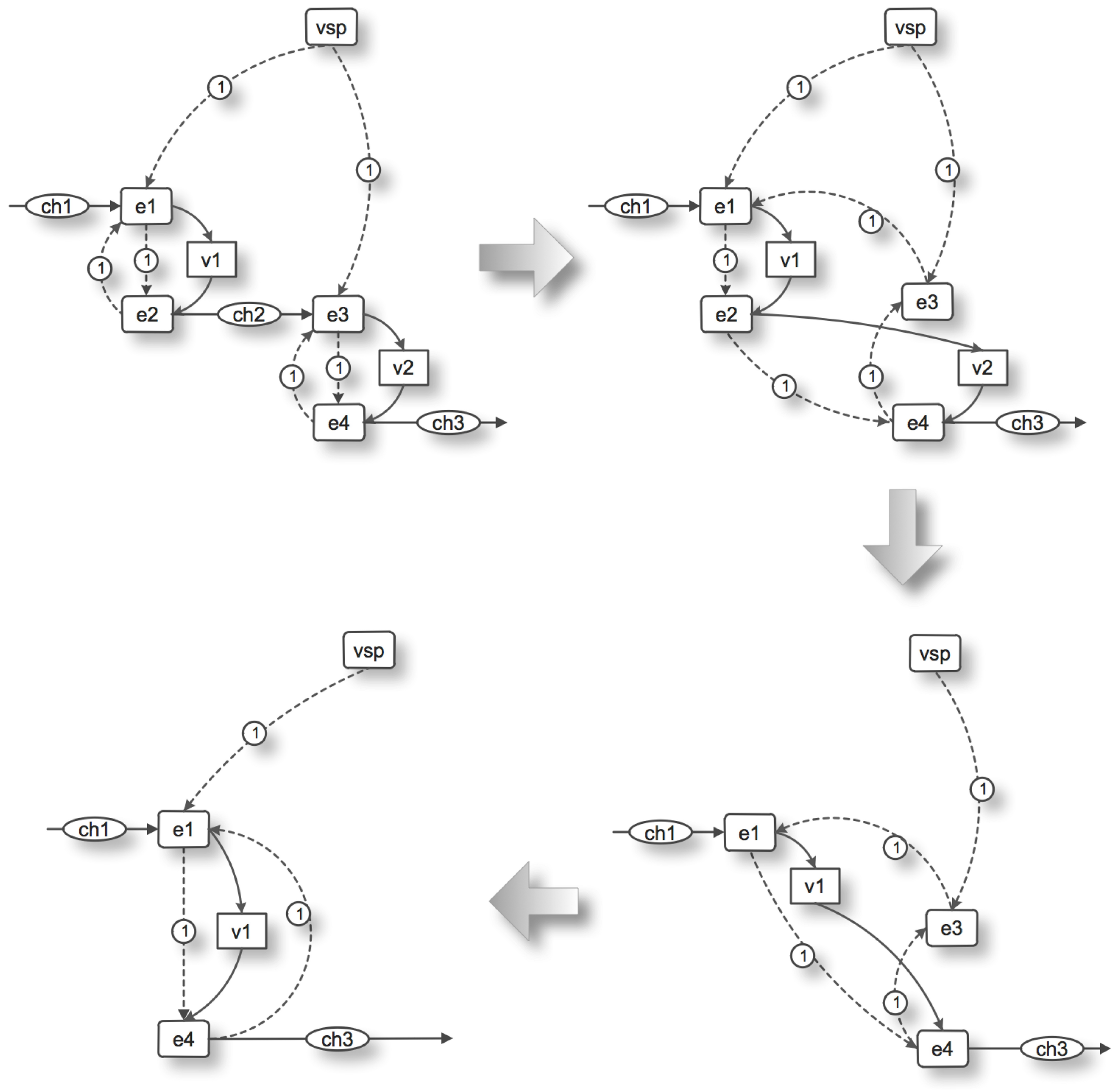


Figure 20: FIFO transformation

Taking several FIFOs with different sizes, we utilized our Verification tool to measure the transformation times. We built FIFOs with sizes 1,2,3,4,5 and 10 places and applied the transformation rules. The results are shown in Figure 21. We can see that, as expected, the total transformation time increases with the size of the FIFO, but still in the order of seconds.

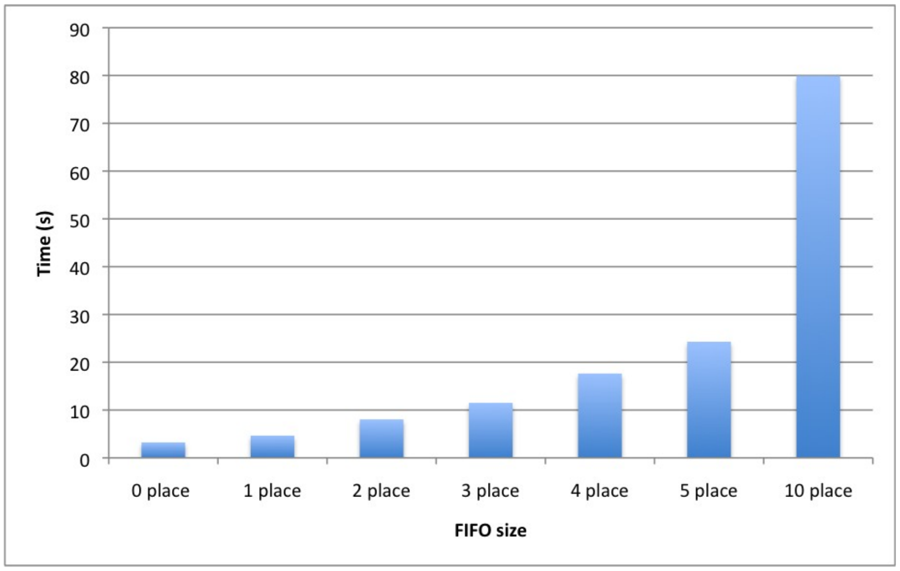


Figure 21: FIFO Transformation times

6.2 Pipeline Modeling

Pipelined architectures are a common optimization for systems with certain types of applications. In a pipeline, data is transferred from one processing module to the next, while all modules run in parallel. This increases dramatically the throughput of the data packets, but the overall time to process one individual packet remains the same. It has been commonly used in microprocessor architecture.

Design decisions

To create an optimum pipeline, the designer must adequately partition the behaviors into different stages. The pipeline throughput depends directly on the slowest module, so the optimum balance in terms of behavior speed must be achieved while partitioning. The pipeline optimization is illustrated in Figure 22. In this figure, we start with 4 sequential behaviors labeled ’B1’, ’B2’, ’B3’ and ’B4’. Their estimated delay times are 5, 10, 20 and 20. The overall delay for each packet would be 55 time units. The optimum way to create the pipeline would be to use 3 pipeline stages, with behaviors ’B1’ and ’B2’ together in the first stage. This way, data packets would be processed every 20 time units.

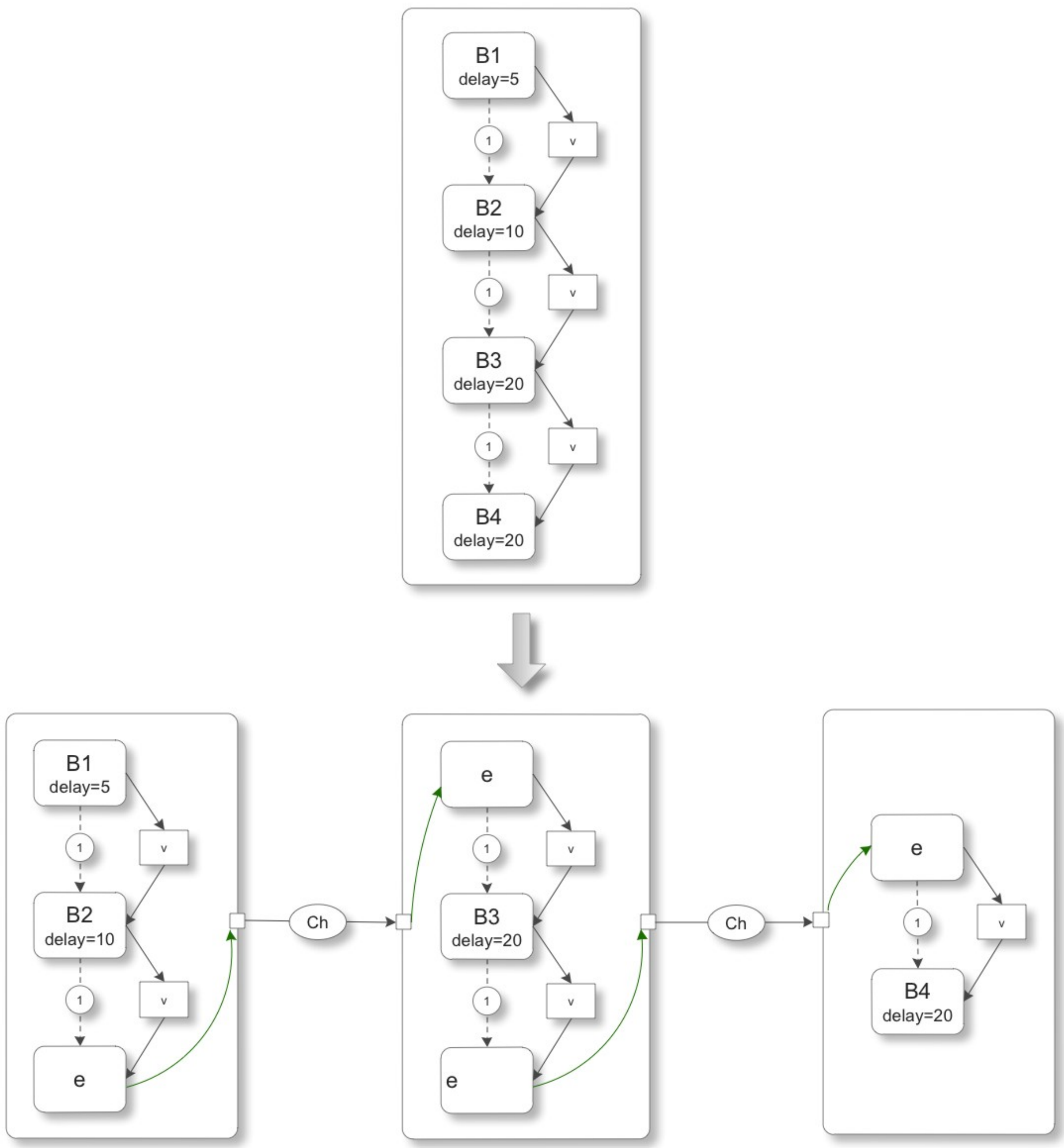


Figure 22: Pipeline optimization

A simple 3 stage pipeline model is shown in Figure 23. We use channels to communicate the data between behaviors *A*, *B* and *C*.

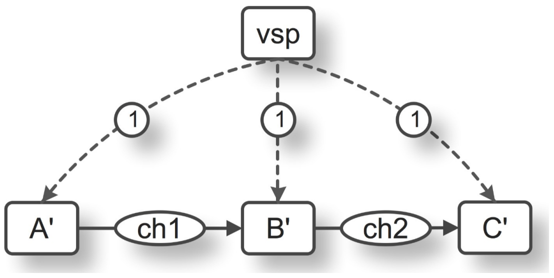


Figure 23: Pipeline Modeling

In Figure 24 we can see the transformation steps to this 3 stage pipelined architecture. Starting in the upper left corner, we can see the model after applying the Flattening rule to the behaviors shown in Figure 23. The next step is to apply the Channel Resolution rule, converting channels *ch1* and *ch2* into control dependencies. The resulting model is shown in the upper right part of the figure. We can see now that behaviors *e2* and *e4* write directly to variables *v2* and *v3*, and both now execute behaviors *e1* and *e3* afterwards. Next, the Redundant Control Dependency Elimination rule is applied and the control dependencies between *vsp* and *e1* and *e3* are deleted. This is shown in the lower right part of the figure. The last step is to apply the Identity Elimination rule to behaviors *e1*, *e2*, *e3* and *e4*. We can see that the model shown in the lower left part of the figure is the representation of a serialized model with behaviors *A*, *B* and *C* executing one after another, using the data written by the previous stage.

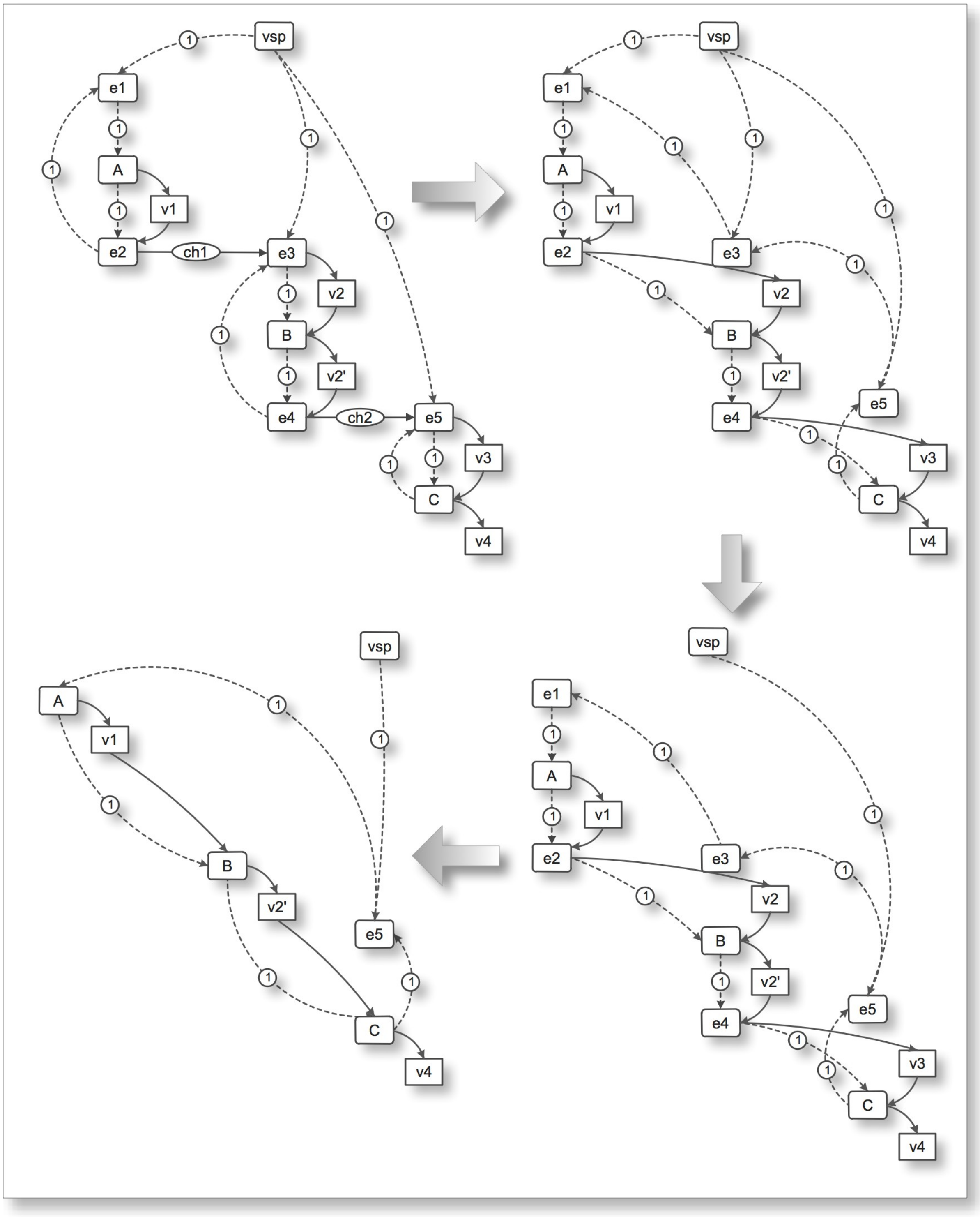


Figure 24: Pipeline Transformations

In order to test our transformation rules, we modeled a JPEG Encoder using Model Algebra's representation. The JPEG encoder is composed of 5 main functions, as shown in Figure 25. They are named: ReadBmp, DCT, Quantize, Zigzag and Huff. They run sequentially in a loop for 180 times, taking a .bmp file as the input and writing out a .jpeg file. Each function can be mapped into a single core or with another function. We created 4 different models, all pipelined, with different number of stages. All were synthesized and implemented into a Xilinx Virtex 4 FPGA board. The mappings for each of the platforms is shown in Figure 26. Each platform is composed of 2 or more MicroBlaze softcore embedded processors, one shared bus (OPB protocol) and one transducer serving as a shared memory. The MicroBlazes will exchange data by storing it in the transducer internal FIFO and reading it out from there.

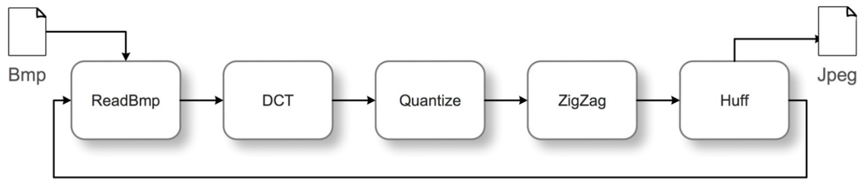


Figure 25: JPEG Encoder Application.

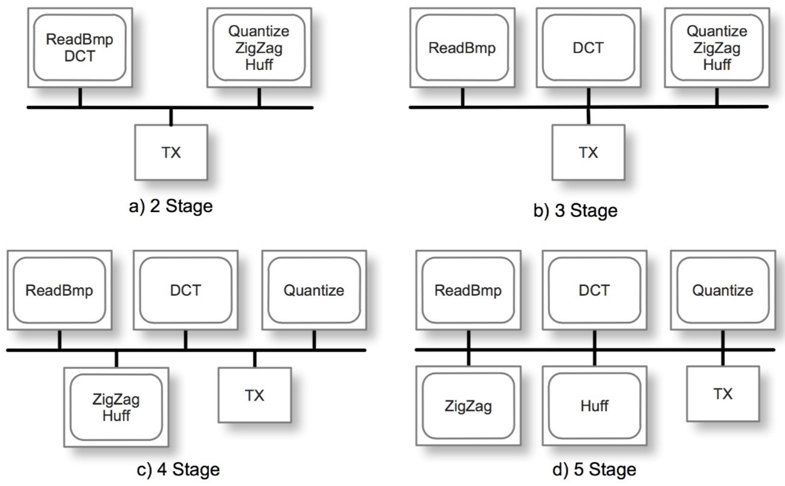


Figure 26: Pipelined JPEG platforms

As shown in Figure 27, the execution time decreases as we increase the number of pipeline stages.

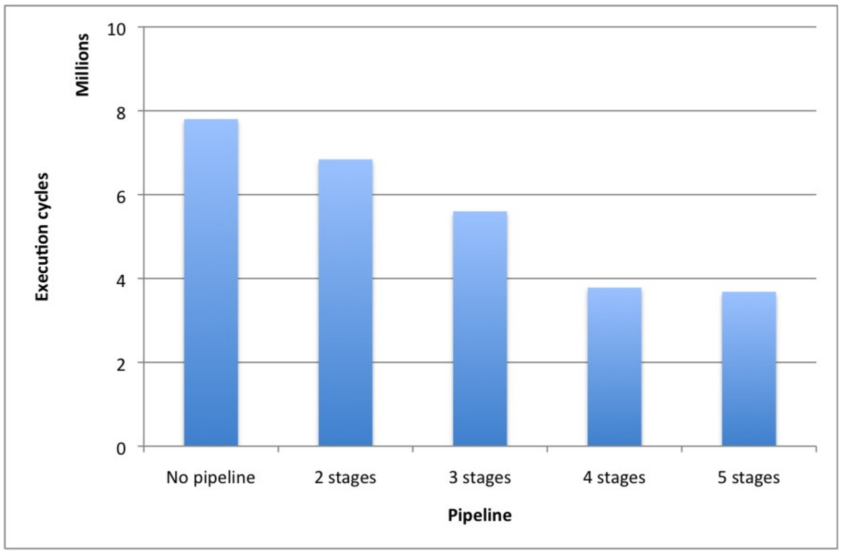


Figure 27: Execution time in a FPGA board

In order to check for equivalence of these different models, we created Model Algebra representations of all 4 and applied the transformations, and checked for isomorphism with the non-pipelined model. The transformation time for these 4 models is shown in Figure 28, and the number and type of transformation rules applied is shown in Figure 29.

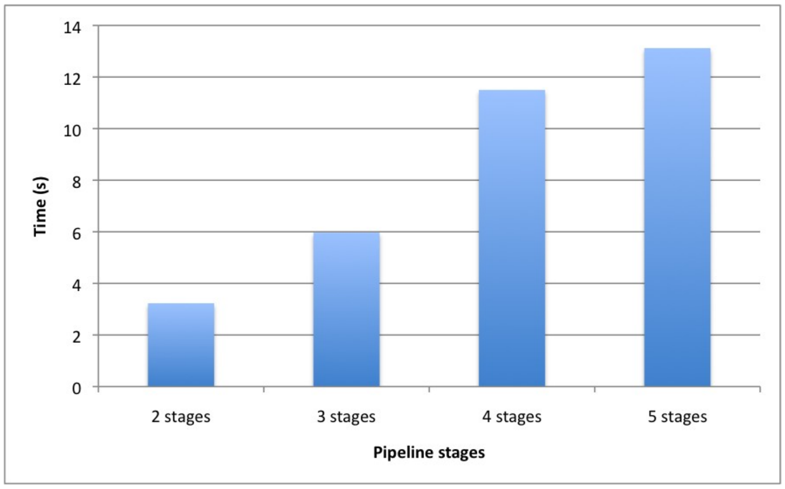


Figure 28: Transformation time in a pipeline architecture

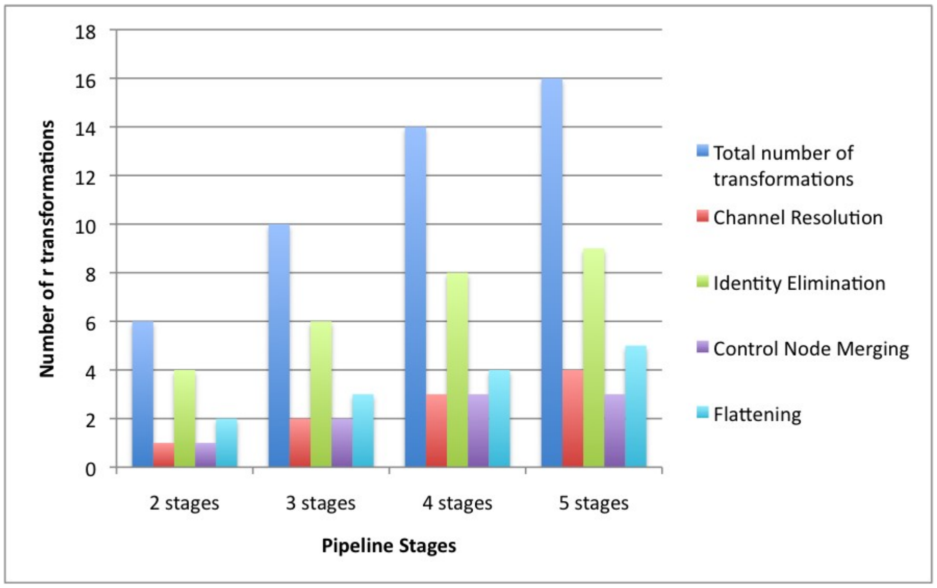


Figure 29: Transformation rules applied to a pipeline architecture

As we can see in Figure 28, the transformation time for each of the platforms is in the order of seconds, and scales linearly. In Figure 29, we can observe that the number of transformations also increases linearly with the number of stages, and all transformation rules applied increase with it.

7 Conclusions

In this article we presented a summary of the system level verification challenges and the Model Algebra formalism. We described new transformation rules for Model Algebra: channel resolution, control node merging and behavior merging. Building upon these new transformation rules, we presented useful system level optimizations, namely pipelining, use of FIFO channels and behavior duplication.

We showed how we could model a N-place FIFO and a pipeline, and used Model Algebra's transformation rules to prove their correctness.

We defined a data structure to describe models written in Model Algebra, named MAG files. This data structure allowed the description, graphical representation and storage of intermediate and final structures of these models in MA.

Using Model Algebra’s composition and transformation rules, we developed a software tool that can take models written in MA (as MAG files), represent them graphically and see transformation rules applied to them. By modeling a multimedia application such as a JPEG encoder into a pipelined architecture, we could prove that Model Algebra’s representation of these optimizations on these platforms could be successfully transformed and compared with the non pipelined model. The transformation time was fast (in the order of seconds) and the number of transformation rules increased linearly with the number of pipeline stages.

The implementation of the software tool gives us more confidence in refinement results, more ability to explore different sequences of transformations and the means to develop correct refinement and optimization tools.

8 References

1. Samar Abdi and Daniel Gajski. Verification of system level model transformations. International Journal of Parallel Programming, 34(1):29-59, February 2006
2. Samar Abdi and Daniel Gajski. A formalism for functionality system level transformations. In Proceedings of the Asia Pacific Design Automation Conference, pages 139-144, 2005
3. Samar Abdi. Functional Verification of System Level Model Refinements. PhD Thesis, University of California, Irvine, 2005.
4. Xml. http://www.w3.org/XML
5. Xml Schema. http://www.w3.org/XML/Schema