Formal Verification for Embedded System Designs

Xi Chen and Harry Hsieh
University of California, Riverside, CA, USA

Felice Balarin and Yosinori Watanabe
Cadence Berkeley Laboratories, Berkeley, CA, USA

Abstract. Embedded electronics today are becoming increasingly complex, which makes their design and analysis more and more difficult. In this paper, we focus on the formal verification of embedded system designs at multiple levels of abstraction, enabled by the Metropolis design environment. Based on the Metropolis framework and the model checker SPIN, a translation mechanism from a Metropolis design to a Promela description is presented and an automatic translator is developed accordingly. We discuss the challenges and solutions in semantically translating from an object-based system design language to a procedural verification language. To demonstrate the correctness and effectiveness of our approach for formal verification, we verify properties for both system level representations and refined representations, where the representations may contain system functions or abstract architectures.

Keywords: formal verification, model checking, Metropolis, Meta-Model, SPIN, LTL

1. Introduction

Electronic products today demand high performance, high integration, and a long list of features. As a consequence, embedded electronics are becoming more complex and difficult to design. To combat complexity and explore design space effectively, it is necessary to represent systems at multiple levels of abstraction. Initial functions and architectures should be specified at a high abstraction level, so as not to bias unnecessarily toward any particular implementation. Through successive refinements and abstractions, the design space can be explored effectively and the design decisions can be made intelligently [12]. Synthesis (i.e. steps taken toward implementation) is applied systematically to transform the specification into manufactured products. Synthesis steps may include structural transformations, where designs are partitioned, composed, or otherwise altered; and formal refinements, where possible behaviors of the design are formally refined through the use of constraints or implementation annotations. A formal grounding for all system representations and operations is essential for the ability to perform analysis and optimization with the high degree of automation.

Our contribution focuses on the formal verification of embedded system designs, especially at higher levels of abstraction. We develop a verification methodology for designs that may go through different levels of abstraction and a translation mechanism from system designs to descriptions more suitable for formal verification engines. We devise solutions to many challenges encountered in semantically translating from an object-based system design language (i.e. Metropolis

© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

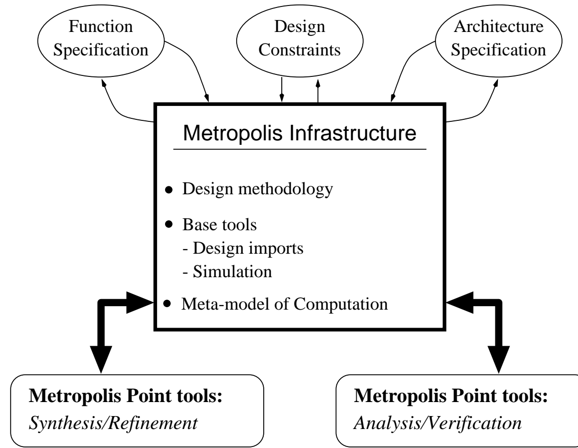

Figure 1. Metropolis design framework.

Meta-Model [5]) to a procedural verification language (i.e. Promela [9]). We demonstrate the correctness and effectiveness of our approach with several verification case studies all using automatic translation.

The Metropolis design framework [5] enables designers to represent and manipulate their designs at multiple levels of abstraction and with multiple models of computation (MoC). Central to the framework is the Metropolis Meta-Model (MMM) representation. Different high-level languages, models of computation, design constraints, as well as specifications of system functions and architecture platforms can be represented in MMM while retaining their correct semantics. Constructs in MMM are chosen to facilitate the transformations and refinements between different abstraction levels. Incorporated into the Metropolis design environment is a set of back-end tools, with which one can simulate, synthesize, and verify the design at hand. This paper describes the formal verification back-end tool. A flow diagram for the Metropolis framework is shown in Figure 1.

Formal property verification can be very powerful for catching errors early in a design process. Formal verification tools, notably model checkers (e.g. SPIN [9] and SMV [15]), are available to designers. One problem for formal property verification is that a verification model needs to be written, often manually, from the specification model. This tedious process multiplies if designers wish to verify the properties of a design as it goes through various abstraction/refinement operations. Metropolis, with its formal semantics, allows full integration of formal verification tools [5]. Verification models can be automatically generated for all abstraction levels of a design, so a designer no longer has to manually re-describe the design in a formal verification language each time the design moves from high levels of abstraction toward implementation. The central challenge in this approach is that the verification languages, such as Promela used by

SPIN model checker [9], allow only simple concurrency modeling and are not amenable to system design specification where complex synchronization and architecture constraints are needed. Our translator automatically constructs the verification model from the specification model, taking care of all the system level constructs. While we focus on the verification methodology for Metropolis designs, the same approach can be easily applied to other abstraction/refinement design frameworks [2].

In the next section, we introduce the basic constructs and semantics of Metropolis Meta-Model and Promela, the input language of the model checker SPIN. In section 3, we discuss several aspects of the translation from Metropolis Meta-Model, an object-oriented system specification language, to Promela, a procedural formal software protocol format [3]. The designer needs to work only at the specification level (i.e. MMM). The constraints, along with the system specification, are translated automatically into the SPIN environment. In section 4, we present our verification methodology and show how verification can be done along with the synthesis procedures. In section 5, we present several case studies of the property verification for system level specifications. Section 6 concludes the paper.

2. Background

In this section, we describe the syntactic and semantic features of Metropolis Meta-Model, the design representation used in Metropolis, and Promela, the input language for SPIN model checker. Despite some apparent similarity between the two languages, MMM contains many system level constructs which need to be translated carefully into semantically equivalent code segments in Promela. The translation mechanism will be described in the next section.

2.1. Metropolis Meta-Model Format

Metropolis Meta-Model is a system representation formalism capable of representing designs at different levels of abstraction. A description of a system (function and/or architecture) can be made in terms of computation, communication, and coordination.

2.1.1. *Processes, Media, and Netlists*

In Metropolis Meta-Model, systems are represented as networks of *processes* that communicate through *media* [5]. Processes and media are used to describe computation and communication respectively. A process is an active object and always defines a function called *thread* as the top-level function where its behavior is specified. A medium implements a set of functions which are grouped into *interfaces*. Processes connect to media through *ports*. Each port has a type, which must be an interface implemented by the medium to which the port is

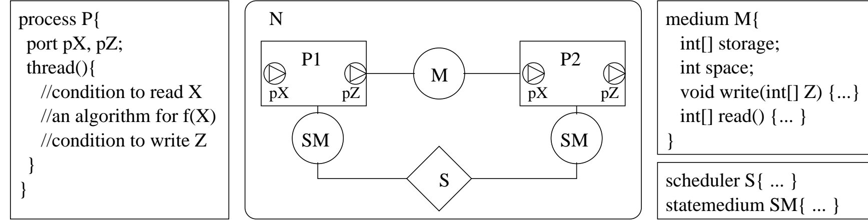

Figure 2. An example of MMM design.

connected. Similar to Java or C++ objects, processes and media have member functions and member variables. Processes communicate to each other through invocation of functions implemented in the shared media. A *scheduler* is a special object of MMM that can be used to specify user-defined scheduling algorithms to control the coordination of a set of processes in a network. Schedulers connect to these processes through *statemedia*, a special type of media. Unlike processes, schedulers are not active objects, which means that they implement functions that are called whenever scheduling needs to be done, but don't have their own running threads. With a properly designed scheduler, the running processes are synchronized in a particular way (e.g. round-robin, priority-based) to satisfy the constraints. In MMM, objects of processes, media, schedulers and their derived types are instantiated in a *netlist*, which can be used to model a complete network. Figure 2 shows an example of a netlist N. The netlist N contains two processes, $P1$ and $P2$, communicating through a medium M. The processes $P1$ and $P2$ are coordinated with each other through a scheduler S and statemedium SMs.

2.1.2. *Await Statement*

Processes run concurrently, each at its own pace. The relative speed of processes may arbitrarily change at any time, unless they synchronize with each other using the synchronization primitive called *await* statement, or some *constraints* are placed on system executions. The await statement can be used to make a process wait until some conditions hold and establish critical sections that guarantee mutual exclusion among different processes. To limit the behavior of processes, a designer can also put high-level LTL (Linear Temporal Logic) [14] or LOC (Logic of Constraints) [4] constraints on the system specification without giving any specific scheduling algorithm, and leave the implementation of these constraints to the detail design stage.

The *await* statement is used to establish mutually exclusive sections and synchronize processes. It contains one or more statements called *critical sections,* each controlled by a triple (*guard*; *testlist*; *setlist*), where *guard* is a Boolean expression, and *testlist* and *setlist* denote sets of interface functions of other processes. A critical section is said to be *enabled* if its *guard* is true, and none of the interface functions in the *testlist* are being executed at that moment. A

critical section may start executing only if it is enabled. In addition, while the critical section is being executed, no interface functions included in the *setlist* can begin their executions. Whenever an *await* is encountered in the execution flow, one and only one of the enabled critical sections is executed. If no critical section is enabled, the execution blocks. If more than one critical sections are enabled, the choice is non-deterministic.

2.2. Promela and SPIN

SPIN [9, 1] is a popular model checker that can be used for the formal verification of distributed software systems, especially protocol designs. In SPIN, the software designs are specified with Promela, a C-style procedural language with a few protocol level extensions. In Promela, the basic concurrent execution units are processes that are defined in *proctype* declarations. The execution of a Promela program always starts from a special process called *init*. If a process is declared as an *active* process, it is instantiated and initiated at the very beginning of the execution by the system. All the other processes need to be instantiated and started explicitly by the existing processes, e.g. the init process. A useful Promela primitive is *atomic*, where only one among all the atomic blocks can be executed at any given time. A communication construct, *channel*, is used to implement both synchronous and asynchronous message passing between processes. The processes may also exchange information through global variables, i.e. shared memory.

As a model checker, SPIN can be used to exhaustively verify a Promela design and prove the validity of any system properties that can be expressed as linear temporal logic (LTL) [14]. Some optimization techniques, e.g. partial order reduction and graph encoding, are available to help reduce the usage of CPU time or memory space. If SPIN cannot complete an exhaustive verification for a complex design with the existing computing resources, one can use the approximate verification, e.g. the bitstate technique [10], to get a partial proof of the property validity. If an error is found during the verification, one can use SPIN's simulator to perform a guided or interactive simulation to find where the error comes from.

3. Translation from MMM to Promela

The main issues in the translation from MMM to Promela include the modeling of MMM processes, media, interfaces, *await* statements (coordinations), and dynamic objects. We do not believe that it will be profitable, at this stage, to develop a new model checker specific to MMM system level specification language. Instead, we rely on automatic translation, both to decouple this very complex problem, and to make it easier for Metropolis environment to take advantage of the latest advancement from the formal verification community.

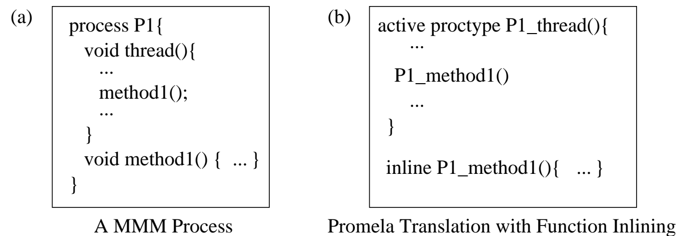

Figure 3. Translation of a MMM process.

3.1. MMM Processes and Media

In MMM, computation is usually modeled as functions defined in processes, and communication between processes is made by calling functions defined in media. We use a translation approach that translates each MMM process instance to a Promela process, and inlines all the functions into the process that calls them directly or indirectly. The translator simply pastes its translated code into the point of the invocation in the calling process. In the situation of multiple level function calls, all the functions are inlined recursively so that one MMM process corresponds to only one Promela process. With function inlining, the verification becomes much more efficient regarding both time and memory usage compared to the dynamic function invocation. Figure 3(a) shows the function $thread()$ of process $P1$ making a call to its member function $method1()$. To perform function inlining, in Promela, the function $P1_method1()$ is declared using the keyword $inline$ as shown in Figure 3(b).

3.2. Interfaces and Await Statements

In MMM, an interface is used to define the I/O data ports of a process or medium and the I/O control points of a process or medium. To implement the control point, an MMM interface is used as a semaphore in the *setlist* and *testlist* of an *await* statement. We translate each interface into a pair of integer variables used as semaphores in Promela. The first variable, called $ACTIVE$ is used to indicate whether the interface (and its member functions) are in active state (i.e. whether they are being executed). Another one called $EXCLUSIVE$ indicates whether this interface semaphore is set (i.e. whether it is included in the *setlist* of some *await* statement that is currently being executed). We use these variables as semaphores to signal that interface functions appearing in *testlist*'s are being executed and to prevent, when appropriate, interface functions appearing in *setlist*'s from being executed. Figure 4 illustrates how an *await* statement is translated to Promela. Promela constructs such as *atomic*, repetition *do-od* and case selection *if-fi* are utilized to guarantee the exact semantics equivalence. Specially, if an *await* statement has more than one critical sections that are enabled, one of them will

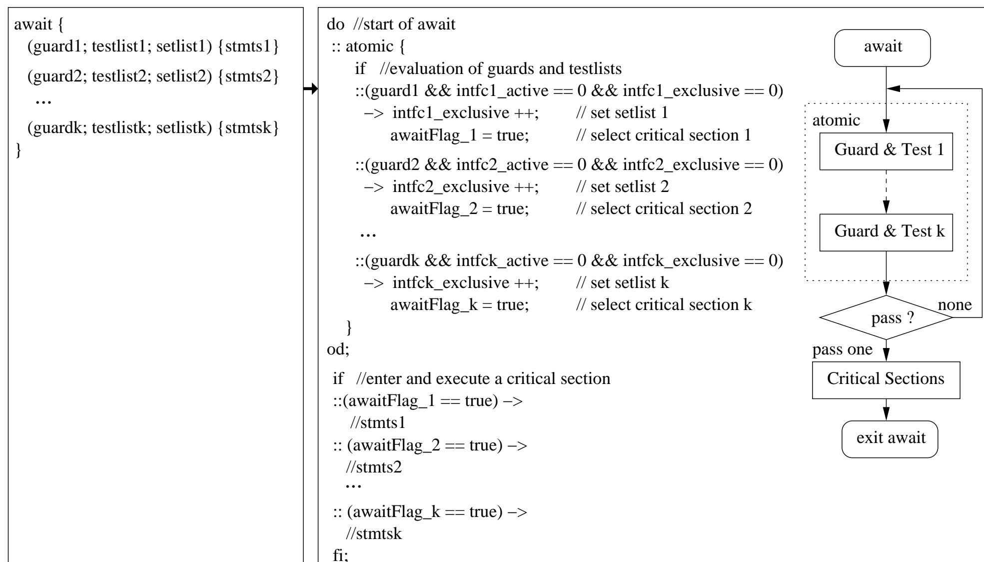

Figure 4. Translation of an *await* statement.

be chosen non-deterministically and executed. This non-determinism is directly supported by Promela in *do-od* and *if-fi* statements.

3.3. Dynamic Objects

Another interesting aspect of MMM is the dynamic object (i.e. the reference type). An array is such a reference type in MMM, and its memory space could be allocated and changed dynamically at runtime. However, most model checkers (including SPIN) only support static memory allocation, i.e. arrays have to be declared explicitly at design time. To solve the problem, we have to put some restrictions on the MMM code. All the reference types have to be declared explicitly once and only once with fixed memory space allocated, so that they can be translated to Promela as static objects. For example, an array in MMM must be declared in the form of "$int[]\ a\ =\ new\ int[12];$", then it can be translated to Promela as a static array "$int\ a[12];$". After an array (e.g. a) is declared in MMM, its reference cannot be changed any more.

4. Formal Verification Methodology

By using an automatic translation procedure to generate a verification model from a system specification model, we allow designers to perform verification at different levels of abstraction as a design goes through various synthesis steps

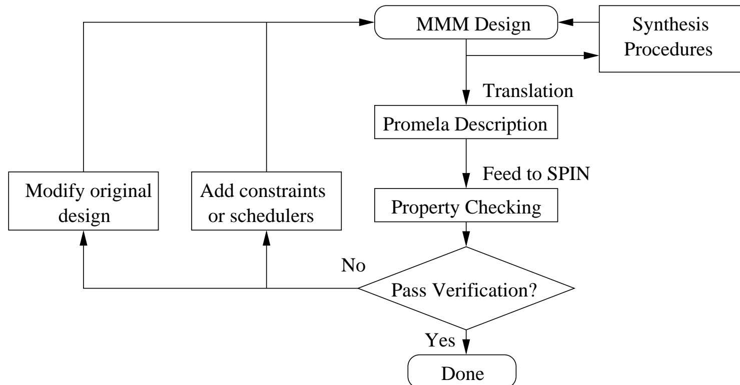

Figure 5. Formal verification methodology.

without the tedious and error-prone step of manually re-writing the design. Furthermore, the verification can be used to drive the refinement and transformation of system representation, i.e. the synthesis procedures.

SPIN provides two powerful ways to specify properties of a design: assertions and Linear Temporal Logic (LTL) [14, 8]. An assertion is an annotation construct in Promela used to "assert" that a particular condition (e.g. space>3) must hold. Assertions written with variables in MMM can be easily translated into Promela segments and inserted into the Promela code after translation. The LTL formulas are constructed using terms, classical boolean operators such as $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\rightarrow$ (imply), and the temporal operators $\Box$ (always), $\Diamond$ (eventually) and $\mathbf{U}$ (strong until). Terms are Boolean conditions on variables or process states. Therefore, LTL is strictly more expressive than assertions. Without loss of generality, we will only deal with LTL formulas in the rest of the presentation.

Our verification methodology is illustrated in Figure 5. A MMM design is automatically translated into a Promela description, and the properties are checked using SPIN model checker. A designer may perform any synthesis step (e.g. composition, decomposition, constraint addition or scheduler assignment) and a new Promela code can be automatically generated to verify the property. If it does not pass, the error trace may be used to help the designer figure out whether the design needs to be altered. If the verification session runs too long, approximate verification can be used to explore a subset of the state space and report the probability that the property will pass. Obviously, a partial exploration can not prove that a property holds. However, it is our experience that a lot of "easy" bugs can be found within a relatively small amount of time and memory usage. If a SPIN verification session continues to run after a long time, it is highly likely that the property will eventually pass.

The same methodology can also be used for a verification-driven synthesis methodology. If the property does not pass the verification, an error trace is

generated and examined. Based on the error trace, the original design may be incorrect, or refinement may be applied to the original specification for it to have the desired property. At a higher level of abstraction, constraints may be used to constrain the behavior so the property may pass. At a lower level of abstraction, schedulers which coordinate the interacting processes may be used to achieve the same result. Subsequent synthesis steps may then actually implement the schedulers on a particular platform.

5. Formal Verification Case Studies

The first set of case studies consider a prototypical network of m producers and n consumers communicating through one or more media (see Figure 6 and 8). The producers receive inputs from the environment, process the data in some way, and then output it to a medium of a single space. The consumers read in information from that medium, process it, and then output to the environment. It is possible for all producers and consumers to execute concurrently. We verify properties of the designs before and after synthesis steps. The second set of case studies involves property checking of systems designed using YAPI model of computation and its more refined, TTL, counterpart. In all the cases, the automatic generation of the verification models (i.e. Promela) from the specification models (i.e. MMM) takes less than one minute of CPU time.

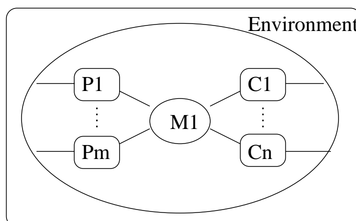

Figure 6. Example of a MMM design.

5.1. Verification of Functional Properties

Given the network of consumers and producers with one medium (see Figure 6), we want to check the property:

"When a consumer wants to read and there is no data in the medium and none of the producers has started to write, the consumer cannot finish reading until some producer starts to write."

In LTL, this property is expressed as:

$$\Box\ (\ (C_x\text{_}start \wedge M_1\text{_}empty \wedge \neg\ (\ P_1\text{_}start \vee \ldots \vee P_m\text{_}start)\ ) \rightarrow ((\neg\ C_x\text{_}end)\ \mathbf{U}\ (P_1\text{_}start \vee \ldots \vee P_m\text{_}start)\ )\ ) \quad , \qquad (1)$$

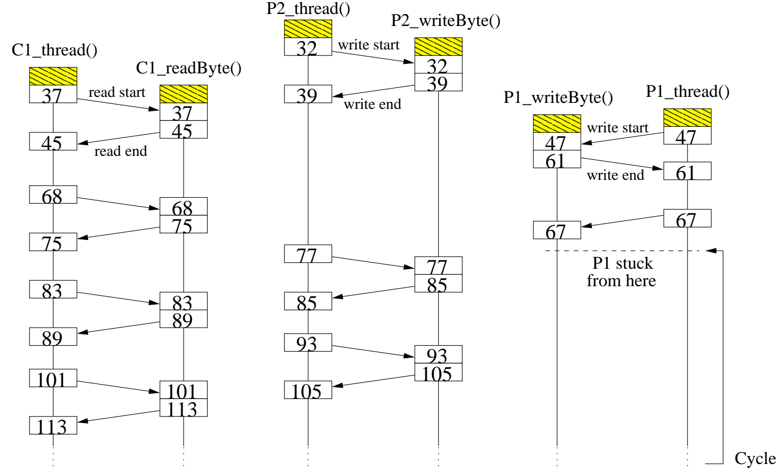

Figure 7. Verification error trace produced by SPIN. The numbers indicate the verification steps and arrows indicate communications between processes through channels.

where *start* indicates the condition that a consumer initiates a read operation or a producer initiates a write operation, and *end* indicates that they complete the operations. We consider the case with m=2 and n=2, which has 102 lines of MMM source code and 670 lines of Promela code after translation. The property is verified by SPIN within one minute of CPU time on our 1.5GHz Athlon machine with 1GByte of memory. The same setup is used for all case studies in this paper. All relevant verification parameters are listed in Table I.

Table I. Summary of verification for property (1).

Search Depth	Total States	State Transitions	Memory	CPU Time
62839	289828	561226	42.6 MB	1.49s

Another property we want to check is:
"If the consumers are able to keep reading data from the medium, then whenever a producer initiates a write, it will eventually complete the write":
In LTL, the property can be expressed as:

$$\Box \Diamond \ (C_1_read \vee \ldots \vee C_n_read) \rightarrow \Box \ (P_x_start \rightarrow \Diamond \ P_x_end) \ , \qquad (2)$$

where *read* indicates the condition that a consumer completes a read operation, *start* indicates the condition that the producer initiates a write operation, and *end* indicates that the producer completes this write operation. Specifically, we prove the case where m=2, n=1 and x=1. SPIN reports that the property does not hold. From the error trace using the debug mode (see Figure 7), we see that

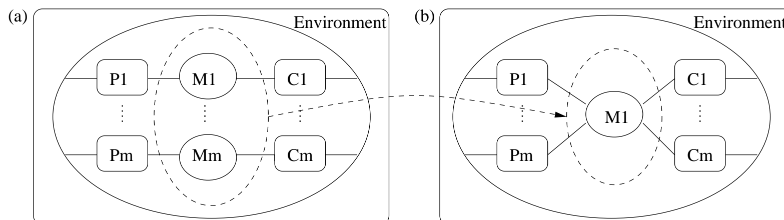

Figure 8. Example of a refinement.

there is possibility of starvation. It is possible for P_2 to keep accessing the medium and prevent P_1 from ever be able to write.

5.2. Constraints and Schedulers

In a Metropolis design, LTL or LOC constraints can be used to limit the possible behavior of a design. However, downstream synthesis procedures must guarantee that the constraints are correctly implemented. If we want the property (2) (with m = 2, n = 1 and x = 1) to hold, i.e. P_1 is not allowed to starve, we may put the following constraint into the design:

$$P_1_start \wedge P_2_start \rightarrow \neg\, P_2_end \; \mathbf{U} \; P_1_end \quad , \tag{3}$$

where *start* indicates the condition that a producer initiates a write operation, and *end* indicates that it completes the write operation. The constraint is trivially "translated" into SPIN environment as the left-hand-side of an implication. Property (2) should be proven only for the cases where the constraint is satisfied. In other words, we prove the LTL formula:

$$\textit{Constraint (3)} \rightarrow \textit{Property (2)} \quad . \tag{4}$$

Formula (4) is proven by SPIN within one minute of CPU time.

In an architecture specification, constraint (3) can be implemented as a scheduler(or arbiter) that has a static-priority policy with P_1 having higher priority. After mapping the function to the architecture, We use SPIN to prove that property (2) (with m = 2, n = 1 and x = 1) holds in the presence of such a scheduler. In addition, if we assign P_2 to have higher priority, the property fails. Another scheduling policy that can be proven to allow property (2) (with m = 2, n = 1 and x = 1 or 2) to hold is the round robin scheduling, where the producers take turns accessing the medium.

5.3. Transformation and Refinement

Of course, system level synthesis procedures may not always be driven by the result of functional verification. For example, communication media may be

combined to reduce the cost. MMM can be used to formally represent the design before and after a particular synthesis step. Consider the example in Figure 8. In (a), m media are used and producer-consumer data streams are running independently. It is trivial to verify that

$$\Box\ (C_x_start \wedge M_x_empty \wedge \neg\ P_x_start \rightarrow \neg\ C_x_end\ \mathbf{U}\ P_x_start)\ , \tag{5}$$

where $x = 1, \ldots, m$, *start* indicates the condition that a consumer initiates a read operation and a producer initiates a write operation, and *end* indicates that the consumer finishes its read operation. Now, let us consider Figure 8(b) where a single medium is used. It is derived from the network in Figure 8(a) through a structural composition. The property

$$\Box\ (C_x_start \wedge M_1_empty \wedge \neg\ P_x_start \rightarrow \neg\ C_x_end\ \mathbf{U}\ P_x_start) \tag{6}$$

is not guaranteed to hold. Indeed, SPIN verifies that the property does not hold within one minute of CPU time. The error trace shows that for the property to hold, a constraint must be added such that streams of data do not mix (i.e. if P_x write, then no other consumer can read until C_x read):

$$\Box\ (\ P_x_write \rightarrow \bigwedge_{y \neq x} (\ \neg\ C_y_read\ \mathbf{U}\ C_x_read)) \quad . \tag{7}$$

With these constraints, the property may be verified by SPIN using the LTL formula:

$$Constraint\ (7) \rightarrow Property\ (6) \quad . \tag{8}$$

We verify the case where m=2. This design has 113 lines of MMM source code and 836 lines of Promela code after translation. The verification completes without error. Table II lists the detailed resource usage of the verification.

Table II. Summary of verification for LTL formula (8).

Search Depth	Total States	State Transitions	Memory	CPU Time
1112111	1.49894e+07	6.6521e+07	101.6MB	31m:54s

We have also run a verification session with a dynamic scheduler of the following form: "if P_x writes, then no other consumer can consume until C_x does". As expected, the property pass with similar complexity measurements. Through experimentation, we have found that no round-robin scheduling nor any static priority real-time scheduler can make the property pass.

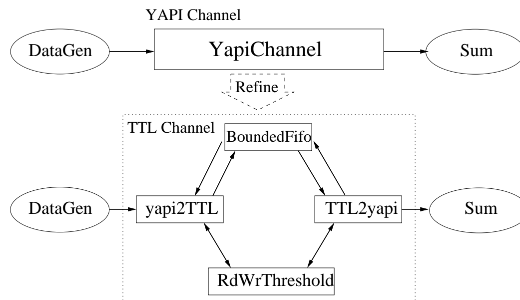

Figure 9. YAPI Channel and TTL Channel.

5.4. YAPI and TTL

Y-chart Application Programming Interface (YAPI) is a popular model of computation for designing signal processing systems [13]. It is basically a Kahn process network [11] extended with the ability to non-deterministically select an input port to consume and an output port to produce. Within Metropolis, a library environment is set up such that any YAPI design can be written using constructs in the Metropolis library. Central to YAPI is the definition of a communication channel and its refinement into Task Transition Level (TTL) [6, 7]. Figure 9 shows how a YAPI channel is refined to a TTL channel in Metropolis. A YAPI channel models an unbounded First-In-First-Out (FIFO) buffer, similar to Kahn process network. Asynchronously, writer processes write data into one end of the channel and reader processes read the data from the other end. At the lower level (TTL), the channel is modeled by a bounded FIFO buffer. The mutual exclusion and boundary checking of the bounded FIFO buffer is guaranteed by a central protocol. As Figure 9 shows, the TTL channel has a bounded FIFO($BoundedFifo$) whose size is set at design time, and a control medium($RdWrThreshold$) which implements a protocol to guarantee correctly writing to and reading from the FIFO buffer. To test the YAPI channel and its TTL refinement, we use a writer process($DataGen$) to write a series of data into the channel and a reader process(Sum) to read the data from the other end of the channel. One property we want to check is that there should be no deadlock situation within the channel, i.e. once the writer starts writing data into the channel, it will finish writing eventually. This property can be expressed as an LTL formula:

$$\Box\ (datagen_start \rightarrow (\Diamond\ datagen_finish))\ . \tag{9}$$

In order to formally verify the YAPI channel, we use a fixed-size array to model its unbounded buffer and choose an array size that is always greater than or equal to the maximum number of elements that need to be stored in the buffer, given the particular processes $DataGen$ and Sum. This property is verified on the YAPI level with exhaustive verification within 9 hours. The YAPI channel

design has 199 lines of MMM source code and 326 lines of Promela code after translation. The other relevant verification parameters are listed in Table III.

We then proceed to verify the TTL channel [6], which has 720 lines of MMM source code and 2158 lines of Promela code after translation. Due to the boundedness of the TTL buffer, the writer will block when there is not enough free buffer slots to write data , and the reader will block when there is not enough data available in the buffer. The TTL channel controller($RdWrThreshold$) implements a protocol that uses a threshold value to indicate if the writer or the reader can be unblocked. If there is a condition on which a process may be unblocked, the controller uses events *wakeup_reader* or *wakeup_writer* to signal unblocking. The detail of this algorithm can be found in [6].

Unexpectedly, the deadlock-free property does not hold any more. In less than 1 minutes of CPU time, the verification fails. After the analysis of the error trace (which is similar to Figure 7), a bug is located in the protocol part of the channel($RdWrThreshold$), where a statement to wakeup the reader is missing. So it becomes possible that the reader is still blocked even when the threshold value indicates that it can be unblocked. After correction (adding one statement to allow the reader to be waken explicitly), we re-run the verification with the existing setup (1.5GHz Atholon and 1GB memory). The verification could not be finished even after it ran for 40 hours and used up the 1GB memory. The refinement process has caused the channel to become much more complex(a protocol controller has been added). We attempt the verification again using the approximate verification, i.e. bitstate technique [10], to verify the property on the TTL channel and get the approximate coverage of at least 98%. Other relevant verification parameters are listed in Table III.

Table III. Summary of verification for YAPI channel and TTL channel.

Design	Search Depth	Total States	State Transitions	Memory	CPU Time
YAPI	19195	4.48827e+07	6.97818e+07	507.5 MB	8h:41m:3s
TTL	43149	1.00611e+08	1.55842e+08	728.5MB	3h:30m:50s

SPIN model checker also provides its own particular mechanism to detect deadlocks. In Promela, designers can explicitly set expected *end* states and let SPIN search the unreachable *end* states that are caused by deadlocks. In our study, we detect deadlocks using an LTL formula, which is functionally equivalent to the *end* state search, but works on the system source code (MMM) rather than Promela. Our deadlock case study is consistent with the idea that designers should work at the system level specification language (MMM) as much as possible so as not to entangle themselves in detailed implementations or the details of the verification models (i.e. Promela).

6. Conclusions

In this paper, we present a verification methodology for system level designs. This methodology is unique in that it is able to operate at different levels of abstraction and allow verification to drive the design process. Integral to the methodology is a semantically correct translator from a system level language, Metropolis Meta-Model, to a software verification language, Promela. Case studies have been performed to show the power of such an approach both in terms of property verification driving synthesis and formal verification of designs before and after a synthesis step. Future work includes verification of complex platform architectures and function-architecture mapping.

References

1. SPIN manual, http://netlib.bell-labs.com/netlib/spin/whatispin.html, 2003.
2. SystemC homepage, http://www.systemc.org, 2003.
3. A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.
4. F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. *International Workshop on High Level Design Validation and Test - HLDVT01*, Sept. 2001.
5. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, pages 45–52, April 2003.
6. J. Brunel, E. A. de Kock, W. M. Kruijtzer, H. J. H. N. Kenter, and W. J. M. Smits. Communication refinement in video systems on chip. *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pages 142–146, 1999.
7. O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. *International Symposium on System Synthesis*, October 2001.
8. P. Godefroid and G. J. Holzmann. On the verification of temporal properties. *Proc. IFIP/WG6.1 Symp. on Protocols Specification, Testing, and Verification*, June 1993.
9. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–258, May 1997.
10. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, 13(3):289–307, Nov. 1998.
11. G. Kahn. The semantics of a simple language for parallel programming. *Proceedings of IFIP Congress 74*, pages 471–475, 1974.
12. K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, Dec. 2000.
13. E. d. Kock, G. Essink, W. Smits, P. v. d. Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: application modeling for signal processing systems. *Proceedings of the 37th Design Automation Conference*, 2000.
14. Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems: Specification. *Springer-Verlag*, 1992.
15. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.