Center for Embedded and Cyber-Physical Systems
University of California, Irvine

Abstracting ESL Designs to UPPAAL System Models

Che-Wei Chang, Rainer Dömer

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

cheweic,doemer@uci.edu
http://www.cecs.uci.edu

Technical Report CECS-14-13
November 21, 2014

Abstracting ESL Designs to UPPAAL System Models

Che-Wei Chang, Rainer Dömer

Technical Report CECS-14-13
November 21, 2014

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

cheweic,doemer@uci.edu
http://www.cecs.uci.edu

Abstract

Formal verification of system level models has been broadly studied to address the completeness concern that the simulation-based validation cannot cover. One approach among formal verification methods is to convert a system-level design into a well-defined representation and make use of existing formal verification tool to analyze the representation along with the properties of interest. In this report, we present an approach to convert an electronic system level (ESL) design in SpecC system level description language (SLDL) into an UPPAAL system level model which is an automaton network for formal verification purpose. Our approach does not only support most of the semantics in the behavioral hierarchy, but also the communication between modules such as event synchronization and most used predefined channels in SpecC semantics. Most important of all, our UPPAAL model can simulate the behaviors of traditional discrete event simulation (DES) and parallel discrete event simulation (PDES). The model can be used for May-Happen-in-Parallel analysis, and for design verification in other aspects, such as timing constraint and power consumption verification.

Contents

List of Figures

Abstracting ESL Designs to UPPAAL System Models

Che-Wei Chang, R Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
cheweic,doemer@uci.edu
http://www.cecs.uci.edu

Abstract

Formal verification of system level models has been broadly studied to address the completeness concern that the simulation-based validation cannot cover. One approach among formal verification methods is to convert a system-level design into a well-defined representation and make use of existing formal verification tool to analyze the representation along with the properties of interest. In this report, we present an approach to convert an electronic system level (ESL) design in SpecC system level description language (SLDL) into an UPPAAL system level model which is an automaton network for formal verification purpose. Our approach does not only support most of the semantics in the behavioral hierarchy, but also the communication between modules such as event synchronization and most used predefined channels in SpecC semantics. Most important of all, our UPPAAL model can simulate the behaviors of traditional discrete event simulation (DES) and parallel discrete event simulation (PDES). The model can be used for May-Happen-in-Parallel analysis, and for design verification in other aspects, such as timing constraint and power consumption verification.

1 Introduction

To cope with the gap between the application in high level language and embedded system platform in hardware description language, System Level Description Language (SLDL) such as SpecC [1] and SystemC [2] are introduced to simplify the design flow and evaluate the software and hardware as a whole at an early design stage. SpecC is one of the SLDL which models the application at higher levels of abstraction and supports implementation features like processing element allocation, module partition, and channel communication. Through the steps of system level design, we verify the functionality along with the implementation details.

Approaches to verify the system-level design can be roughly divided into two sub-categories: 1) simulation-based validation and 2) formal analysis and verification. The advantage of the simulation-based validation is that it is fast compared to the formal approaches, but the disadvantage is that it cannot prove that the completeness, i.e., it cannot guarantee if a certain property is satisfi-

able or unsatisfiable under all possible inputs and conditions. To address the completeness concern in the verification, formal approaches are widely studies to prove or disprove the satisfiability of the property of interest. A widely used approach in formal verification is to convert the system level design into a well-defined representation and then make use of existing formal verification tools to analyze and verify the representation. In this report, we propose an approach to convert the system design in SpecC SLDL into an UPPAAL [3] system model in the form of automaton network so that we can use UPPAAL model checker to analyze and verify the model. There are also other works using UPPAAL model checker to formally verify the system-level design in SystemC [4]. Compared with the other work using UPPAAL model checker, our approach generates an UPPAAL model which does not only simulate the behavior of traditional discrete event simulaiton, but also parallel discrete event simulation (PDES) [5] in which concurrent threads are executed by processor cores in parallel.
This report is organized as follows: In Section II we briefly introduce the UPPAAL system model and in Section III we show the conversion from system level design in SpecC SLDL to UPPAAL system model. We conclude this work and its future applications in Section IV.

2 UPPAAL System Model

Before the description of our approach, we first briefly introduce the basic concept of an UPPAAL system model. An UPPAAL model consists of a network of concurrent processes which are created by instantiating the pre-defined timed automaton templates, and these concurrent processes can communicate and synchronize with each other through parameters and channels defined. The system can be seen as a set of automata running concurrently, i.e., when there are multiple transitions enabled in the instance processes, these enabled transitions can take place in non-deterministic order. An UPPAAL system model is usually composed of three parts:
1) definition of data structures, functions and global variables declaration,
2) definition of automaton templates, and
3) system definition.
The first part is quite similar to programming language like C. In an UPPAAL model the designers can define global vairalbes and function to be accessed and called by all instance processes. Except for the basic variable types supported by UPPAAL modeling language such as integer and Boolean variable, designers can also define their own complex data structures using *struct* construct. As for the second and third parts, we use a simple model in Fig. 1 to illustrate the basic components in an UPPAAL model. In this example, the model is composed of two processes `Inst1` and `Inst2` (illustrated as green blocks in the figure) communicating through channel [sync] and integer [a].

The templates of the automata has to be defined first and then they can be instantiated in the system definition to create the processes and build the model. To build a model in Fig. 1, automaton template `TA1` and `TA1` have to be defined first. In the definition of a template, states in the automaton, transitions between states, the conditions to enable a transition and expression to be evaluated on the transition are clearly specified. In UPPAAL model they are named as *location*, *transition*, and *label* respectively. Take template TA1 as the example. Four locations X1$\sim$ X4, transitions X1→X2, X1→X3, X2→X4, and X3→X4 are defined. Labels are shown as blocks on transitions in the illustration, and they are attached to transitions to specify the expressions and conditions in which transitions are enabled. UPPAAL model checker supports three types of label for different purposes. The first type are *update* labels (b=1, b=3, or a=b+1 in black in this example) define the expression to be evaluated during the transition. The second type are *guard* labels represent the condition when transitions are enabled. When process Inst2 is at location Y2, integer *a* defines which transition is enabled in this process. Note that when *a* equals 3, process Inst2 stays at location Y2. The third type are *synchronisation* labels which define the event synchronization between transitions in multiple processes. The synchronisation labels with exclamation mark are event producers and the labels with question marks are consumers. In this example, whenever transition X2→X4 or X3→X4 happens, the transition from Y1→Y2 happens at the same time if process Inst2 is at location Y2.

The final step to build an UPPAAL model is to instantiate predefined templates and create a network with concurrent processes in the system definition. The instance processes created in the system definition can communicate with parameter and channel. In this example, channel [sync] and integer [a] are defined in the system definition and used to connect processes Inst1 and Inst2.

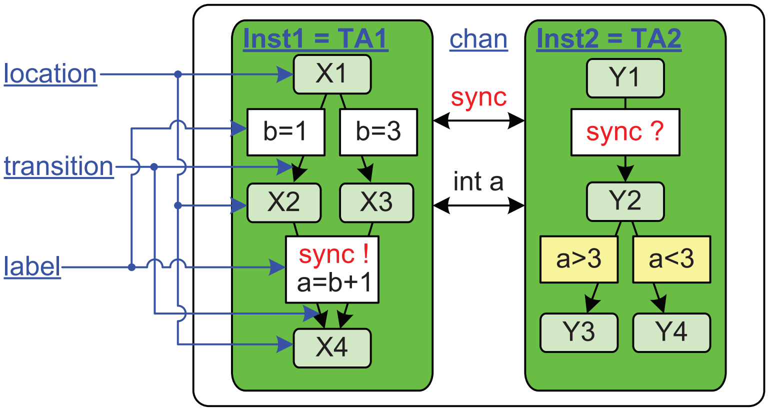


Figure 1: Example of an UPPAAL system model

3 SLDL Design to UPPAAL Model

In this section, we first briefly introduce the basic concept of mapping a system design into an UPPAAL system model. After the introduction, we describe how we convert the details of behaviors and scheduler for parallel discrete event simulation algorithm into UPPAAL automaton templates.

3.1 PDES model in UPPAAL

Fig. 2 shows our structure of the UPPAAL model for a system model. A system model is usually composed of multiple computation blocks(modules, behaviors) with communication (port, channel, event synchronization) between those blocks. We distinguish two types of behaviors: *Leaf* and

Hierarchical behavior, which implements the computation and specifies the composition of leaf instances, respectively. A system model is constructed with a topmost behavior `Main` and the sub-instances of hierarchical and leaf behaviors.

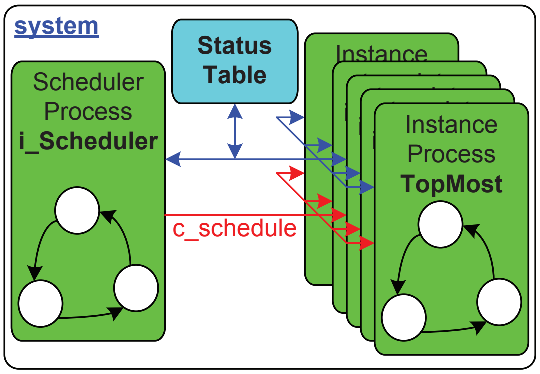
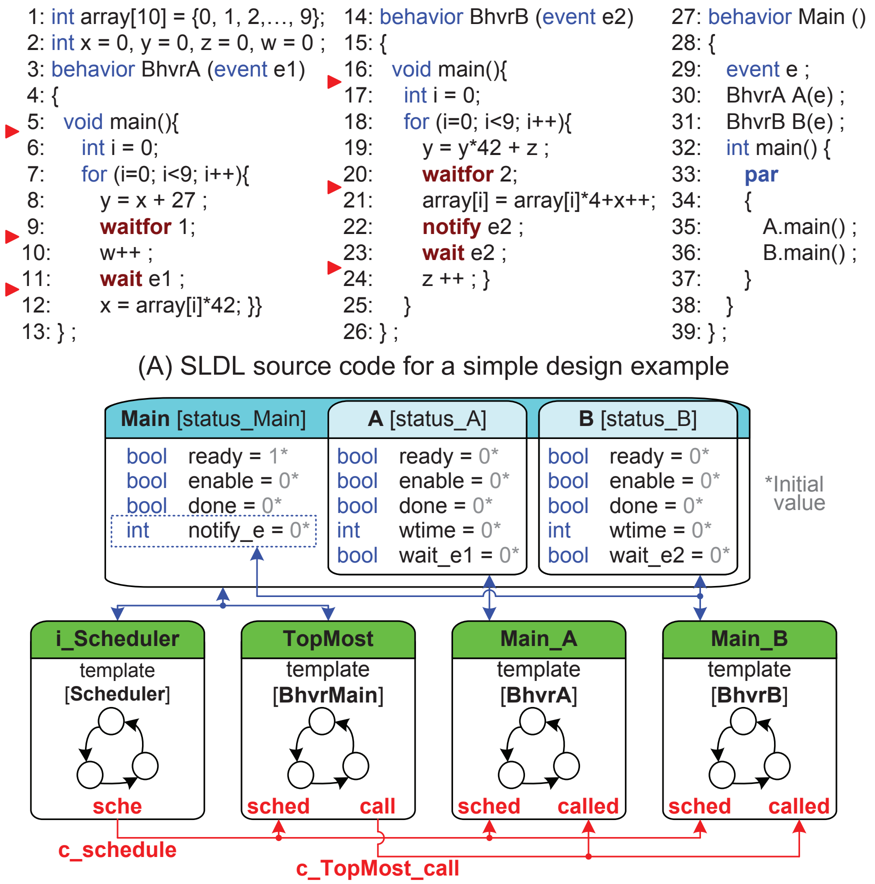


Figure 2: SLDL Design to UPPAAL automata conversion

In our approach, we first abstract an automaton template from each behavior. These templates are then instantiated to build a process network modeling the system. Each behavior instance in the design is one-to-one mapped to a process through template instantiation. The system also contains a *scheduler* process to coordinate the transitions in instance processes. All instance processes are connected to the scheduler process through a structure *status_tree* and a channel *c_schedule*. The status tree is a tree structure designed to keep the status information for all behavior instances. The status flags *ready*, *enable*, and *done*, are kept in the node to represent the status of the corresponding instance, and certain additional flags such as *wtime* or *notify_X* are added to the node according to the statement in the behavior. Based on the information in [status_tree], the scheduler activates instance processes in the proper order and ensures the transitions are compliant with the parallel discrete event execution semantics.

We provide an introductory SLDL example from [6] in Fig. 3 to demonstrate the structure of our UPPAAL model as well as the [status_tree]. As shown in Fig. 3, four processes are created through template instantiation in the system definition for the scheduler, topmost behavior Main, and its child instance A and B, respectively. All instance processes are connected to the scheduler process through [c_schedule] and [status_tree]. Note that there is also a channel *c_call* between the parent process and its child processes, as in some hierarchical behavior the child instance are activated by the parent but not the scheduler.

While the hierarchy of instances has been flattened in the system definition of the UPPAAL model, [status_tree] still maintains the hierarchy of the design. The reference of each node is passed to the corresponding instance process as parameter so that the process can access its flags and its children's. Except for the reference of the node, the reference of a flag can also be passed to a process as needed. Take flag [notify_e] as an example. The reference of this flag is passed to process Main_B for statement `"notify e2"`in BhvrB as process Main_B needs to set this flag when it reaches the location of the notify statement.



```
int array[10] = {0, 1, 2,…, 9};
int x = 0, y = 0, z = 0, w = 0 ;
behavior BhvrA (event e1)
{
  void main(){
    int i = 0;
    for (i=0; i<9; i++){
      y = x + 27 ;
      waitfor 1;
      w++ ;
      wait e1 ;
      x = array[i]*42; }}
} ;

behavior BhvrB (event e2)
{
  void main(){
    int i = 0;
    for (i=0; i<9; i++){
      y = y*42 + z ;
      waitfor 2;
      array[i] = array[i]*4+x++;
      notify e2 ;
      wait e2 ;
      z ++ ; }
  }
} ;

behavior Main ()
{
  event e ;
  BhvrA A(e) ;
  BhvrB B(e) ;
  int main() {
    par
    {
      A.main() ;
      B.main() ;
    }
  }
} ;
```

(A) SLDL source code for a simple design example

(B) UPPAAL model for the simple design example

Figure 3: SLDL source code for an introductory design example

3.2 Automaton Template for Hierarchical Behaviors

Fig. 4 shows automaton template for a hierarchical behavior in our approach. The left-hand side of the illustration shows the basic structure for all types of behaviors. In each behavior template there are at least three locations: [Idle], [Initial], and [End]. All behavior processes start at [Idle], and wait for transition [Idle]→[Initial] to be activated. Transition [Idle]→[Initial] is only activated when the enable flag in the status node for the corresponding instance is set and the synchronization is triggered through channel [c_schedule] or [c_call]. After the execution of the behavior instance is finished, the process reaches [End] and then goes back to [Idle]. Fig. 4 also shows the locations and transitions for the four types of composition defined in SpecC SLDL.

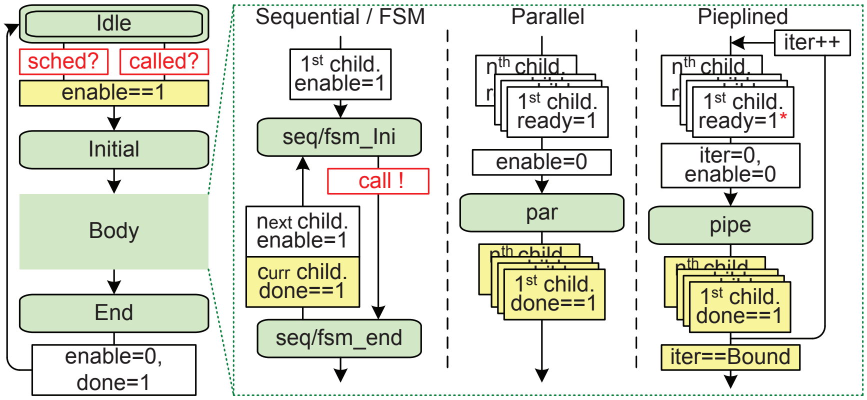


Figure 4: Representation of Hierarchical behaviors in UPPAAL

3.2.1 Sequential and FSM Composition

For both sequential and finite-state-machine (FSM) composition, the instances are executed sequentially and only one instance is activated at a time. In a hierarchical behavior of sequential composition, the children instances are executed in the order in which they are instantiated. When the parent process reaches location [seq_ini], the enable flag of the first child instance is set by the parent process, and the parent process activates the transition [Idle]→[Initial] in child by triggering the synchronization over channel [c_call] with synchronize label in transition [seq_ini]→[seq_end]. After the activated child process reaches [End], the done flag of the child process is set and transition [seq_end]→[seq_ini] is enabled. The parent process then activates the next child process in the same manner. After the execution of all children finishes, transition [seq_end]→[End] is enabled, and the parent process sets its done flag, resets the enable flag, and goes back to [Idle]. For the FSM composition, the interaction between parent and child instances are similar to the sequential composition, but the child automata are activated in the order specified in the FSM transition statements.

3.2.2 Parallel and Pipelined Composition

The instances in a behavior with parallel or pipelined composition are executed concurrently. To model the parallel execution semantics, all child processes are activated at the same time in the UPPAAL model. As shown in Fig. 4, in transition [Initial]→[par] the update label sets the ready flags of all child instances and clears enable flag of the parent instance. The scheduler process detects the assertion of the ready flags and activates child processes at the same time by setting the enable flags of all child instances and triggering the synchronization over channel [c_schedule]. The parent process waits at location [par] until the done flags of all child instances are set.

As for the pipelined composition, considering the filling and flushing stage, not all child instances are activated in all iterations. For a pipeline composition with n instances and m iterations, the ready and done flag of i-th instance at s-th iteration, $i \in \{1,2,...,n\}, s \in \{1,2,...,m\}$, are set in the iterative transition [pipe]→[pipe] as follows:

$$\begin{array}{ll} \mathtt{if}\ \mathtt{i} \le \mathtt{s} \le \mathtt{m}+\mathtt{i}-1, & \mathtt{Inst(i).ready}=1, \mathtt{Inst(i).done}=0 \\ \mathtt{else} & \mathtt{Inst(i).ready}=0, \mathtt{Inst(i).done}=1 \end{array}$$

3.3 Automaton Template for Leaf Behaviors

As for the abstraction of leaf behaviors, instead of generating location and transition for every statements, only certain statements of interest are taken into consideration in the model generation. Here we categorize the statements of interest into three types: 1) control-flow statement, including if/if-else, while/do-while and for loop, 2) time advancement and event synchronization, and 3) channel communication. Statements other than these three types are abstracted away (ignored) since they have no influence on the transition in the automaton. Note that if there is no waitfor, wait-notify statement or channel communication in the sub-statements, the control-flow statement is abstracted away, too.

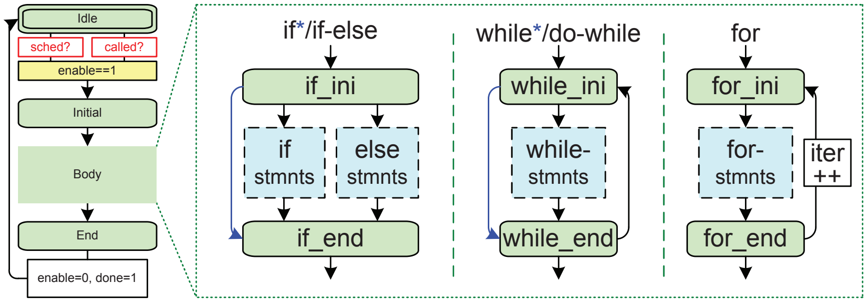


Figure 5: Control flow statements reflected in leaf automata

3.3.1 Control-flow Statement

Fig 5 shows the corresponding locations and transitions generated for if/if-else, while/do-while, and for loop. We generate a pair of locations [ini] and [end] for these three types of statements to encapsulate their sub-statements. For if/if-else statements, we create transitions from location [ini] into the sub-statements for both cases, and two paths merge at location [end]. For the do-while/while loop statement, transition [end]→[ini] is inserted to execute the substatement for non-deterministic times, and a transition bypassing the substatement is provided for the while statement in case the condition is false at the first iteration. The for-loop statement is similar to the while-loop with guard and update labels in the transition to count the iteration.

3.3.2 Time Advancement and Event Synchronization

The time advancement and event synchronization in our system model is implemented with waitfor and wait-notify statements. The locations and transitions inserted for waitfor, wait, and notify statements are illustrated in Fig. 6. Two locations [ini] and [end] are created for each wait and waitfor statement, and one location [notify] is inserted for each notify statement. According to the execution semantics, it takes at least one delta cycle or simulation clock advancement to wake up an automaton from suspension caused by wait or waitfor. Since the scheduler process is the only module aware of time advancement, the suspended automata are re-activated by the scheduler.

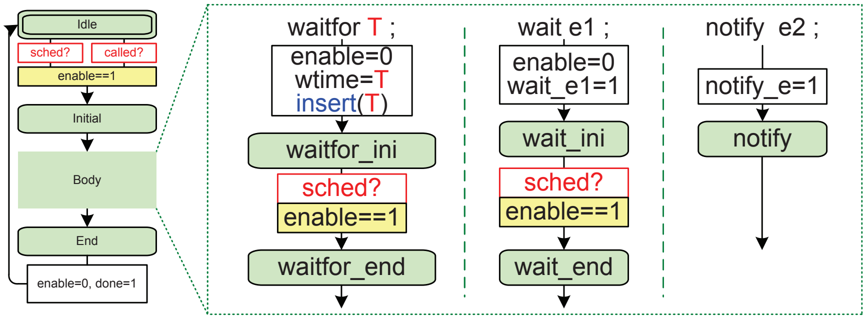


Figure 6: Waitfor statement and wait-notify synchronization

A waitfor statement with argument T suspends the current instance from execution for T time units. The suspended process will be wakened up by the scheduler processes after the simulation clock is advanced by N time units. In our UPPAAL model, a global sorted queue is used to store the waiting time of all suspended instances, and flag [wtime] in the status node also keeps the remaining waiting time for the corresponding instance. When a process reaches location [ini] of statement `waitfor T`, it suspends itself and set its flag [wtime] to `T`. A predefined function `insert` is also called to insert `T` into a global sorted queue for time advance in the scheduler. The waiting time in the queue will be read out in order in the scheduler process to decide what is the next time units to be advanced. After the simulation time is advanced by `T` units and flag [wtime] of the suspended instance is reduced to zero, the process is reactivated by the scheduler.

A wait statement suspends the current thread from execution and waits for a statement notifying the same event is executed. In our model, a *wait* flag is added to the status node for each event argument in the instance. The flag is set to specify the corresponding instance is waiting for event delivery. Take the introductory design in Fig 3 as the example. Flag [wait_e1] and [wait_e2] are added to the status node of instance A and B for argument `event e1` and `event e2`. When a process reaches location [ini] of a wait statement, it suspends itself and sets the corresponding wait flag. When any process reaches the location of a notify statement notifying the event, the suspended process is reactivated by the scheduler.

A notify statement wakes up all suspended threads waiting for the notification of a certain event. In our model, a *notify* flag is added to the status node for each event. The reference of the flag is passed to all instances notifying the event so that those processes can set the flag when they reaches the notify location. For example, in Fig 3 flag *notify_e* is added to the status node of Main, and the reference of this flag is passed to the process of instance B for the statement `notify e2`at line 22.

3.3.3 Channel Communication

Channel communication is essential in system level modeling, and SpecC SLDL supports various standard channels, such as semaphore, mutex, handshake, double-handshake, and queue. In SLDLs, the channel communication between blocks is implemented by making function calls to the method defined in the channel instances to transfer data from sender to receiver. Our approach supports the modeling of the three mostly used channels, which are handshake, double-handshake, and queue. We show the standard double handshake channel illustrated in Fig. 7 as the example. In this example, channel instance C is connected to instances S and R so that these two instances can call functions send() and receive() defined in the channel to communicate.

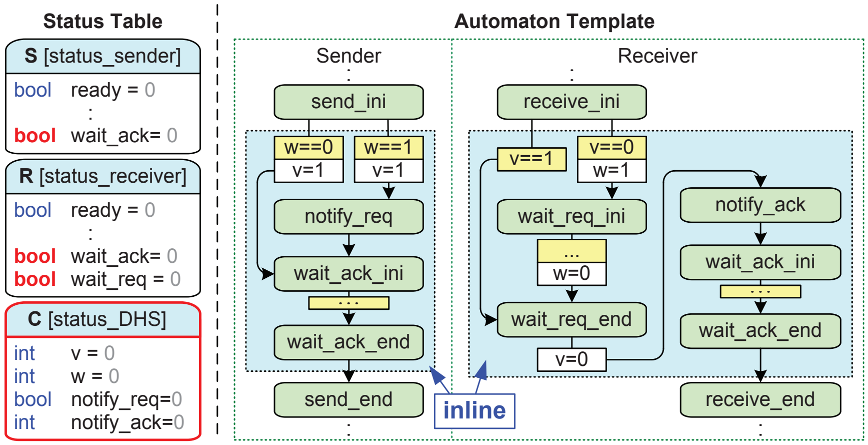


Figure 7: Communication using standard double handshake channel

In the UPPAAL model, we inline the predefined communication method into the sender and receiver process. The text and block in red in the left part of Fig. 7 shows the additional node and flags for using a double handshake channel. The right part illustrates the inlining of locations and transitions modeling the detail of the communication with a double-handshake channel. Except for locations [ini] and [end] inserted for the channel function call, The locations and transitions modeling the detail of the communication method are also inlined between [ini] and [end]. Flags *wait_ack* and *wait_req* are added to the status node of the sender S and receiver R, and a status node C is inserted in the tree for the channel instance C. The wait and notify locations here synchronize the sender and receiver, and the guard and update labels in the transitions make sure the send and receive function finish at the same time.

3.4 Scheduler Automaton

In this section we show the scheduler process modeling the discrete event simulation. Note that our scheduler automaton supports the modeling for both regular DES and parallel DES. The difference is that the regular DES mode allows one active process at a time, while the parallel DES mode allows many activated processes. Since in this paper the scheduler needs to run in PDES mode for MHP analysis, the following description of scheduler automaton is for PDES mode. Fig. 8 illustrates the template of the scheduler automaton. The composition of the scheduler automaton can be roughly divided into three parts: 1) instance activation, 2) event delivery, and 3) time advancement.

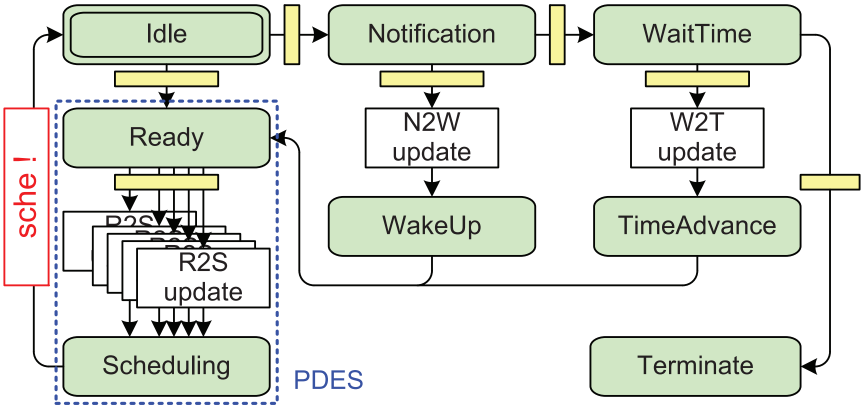


Figure 8: Scheduler automaton with delta and time advance cycles

The instance activation contains the loop from [Idle] to [Ready] and [Scheduling], and then back to [Idle]. Transition [Idle]→[Ready] is enabled when there is any asserted ready flag in the status tree. The enable flags of all instances with asserted ready flags are set in transition [Ready]→[Scheduling], and the all instances with asserted enable flag are activated by the synchronisation label in transition [Scheduling]→[Idle].

The event delivery includes the path from [Idle] to [Ready] via [Notification] and [WakeUp]. This part simulates the delta cycle increment in the DES. Transition [Idle]→[Notification] is enabled when all instances are suspended. The guard in transition [Notification]→[WakeUp] checks if there is any asserted notify flag, and the update label sets the ready flags of the suspended instances waiting for the same event. For example, the labels below are annotated to the transition to wake up instance A and B in Fig. 3 from suspension.

$$\begin{aligned} [\textbf{guard}] &\ \ \texttt{Main.notify\_e} == 1 \\ [\textbf{update}] &\ \ \texttt{Main.A.ready} = (\texttt{Main.A.wait\_e1} == 1)?\, 1 : 0, \\ &\ \ \texttt{Main.B.ready} = (\texttt{Main.B.wait\_e2} == 1)?\, 1 : 0 \end{aligned}$$

The time advancement is the path from [Idle] to [Ready] via [WaitTime] and [TimeAdvance]. This part simulates the simulation time advancement in the DES. Transition [Notification]→[WaitTime] is enabled when there is no asserted notify flag in the status tree. The guard in transition [WaitTime]→[TimeAdvance] reads the minimal waiting time *min_clk* from the sorted queue and advance the time by *min_clk*. If min_clk is 0, i.e., there is no instance waiting for time advancement, the transition to [Terminate] is enabled and the scheduler process can end. If the minimal time value is greater than 0, the update label in transition [WaitTime]→[TimeAdvance] set the ready flag of the suspended instance if its wtime flag matches min_clk. The following labels are annotated to transition [WaitTime]→[TimeAdvance] to wake up instance A and B from suspension in Fig. 3.

$$\begin{aligned}[\textbf{guard}]\ & \texttt{min\_clk} > 0 \\ [\textbf{update}]\ & \texttt{Main.A.ready} = (\texttt{Main.A.wtime} == \texttt{min\_clk})?\ 1:0, \\ & \texttt{Main.B.ready} = (\texttt{Main.B.wtime} == \texttt{min\_clk})?\ 1:0\end{aligned}$$

min_clk will then be subtracted from all wtime flags greater than 0 in the status tree as well as from all waiting times in the sorted queue.

3.5 UPPAAL System Description for a PDES Model

After automaton templates for behaviors and centural scheduler are defined, the last step is to instantiate the defined templates in the system description to build the model. As we described before, each instance in the design is one-to-one mapped to a instance process in the system description. Our approach flattens the hierarchy of the system, and we rely on the synchronization channels between scheduler and instance processes as well as the channels between parent and child instances processes to coordinate the transitions in these concurrent processes and simulate the execution semantics defined in SpecC language. Here we use the introductory example in Fig. 3 to demonstrate the generated system description. An UPPAAL system model illustrated in Fig. 9 is generate for the introductory example. For simplicity, the labels on the transition and communication between processes are not shown in the figure.

4 Conclusion and Future Work

In this work, we propose an approach to model a system-level design in SpecC SLDL as an UPPAAL model. Our method covers most of the compositions in hierarchical behaviors including sequential, parallel, pipeline and fsm, and support control-flow statement, event synchronization and time advancement statement in leaf behaviors. As for the channel communication, our model also supports three most used channels which include handshake, double-handshake, and queue channel with arbitrary buffer size. Most important of all, our model simulate both regular discrete event simulation and parallel discrete event simulation.

There are many possible applications we can apply this apporach to. Based on this model, we can formally verify properties of interest in the system-level design, such as potential deadlock detection, timing/power constraint verification, and May-Happen-in-Parallel analysis. We also plan to use this model to optimize the power consumption of a system.

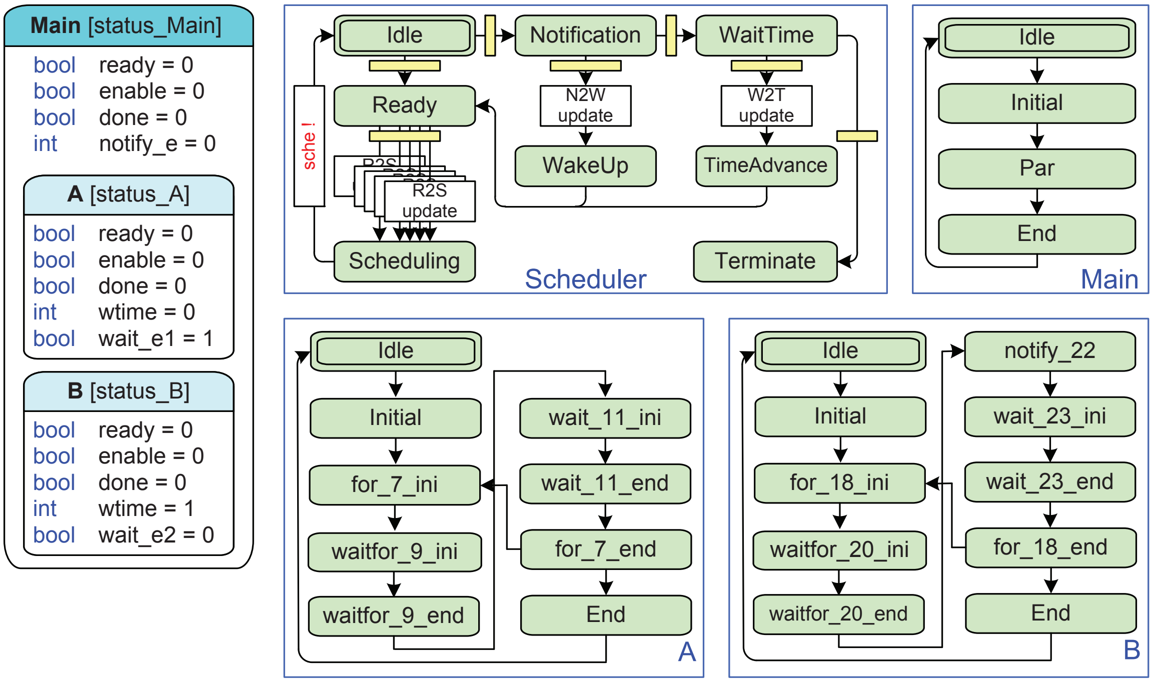


Figure 9: UPPAAL system description for the introductory example

References

[1] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[2] IEEE Standards Association, *"IEEE Std. 1666V2005, Open SystemC Language Reference Manual,"* 2005

[3] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P.Pettersson, W. Wi, and M. Hendrikes, *"UPPAAL 4.0,"* in Proc. QEST, 2006, pp. 125-126.

[4] P. Herber, and S. Glesner, *"A HW/SW co-Verification framework for SystemC,"* in ACM Trans. Embed Comput, Syst. 12, 1s Article 61, 2013.

[5] W. Chen, X. Han, and R. Dömer, *"Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment,"* in IEEE Design and Test of Computer, vol. 28, pp. 20-31, 2011.

[6] W. Chen, X. Han, and R. Dömer, *"May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models,"* in DATE '14 European Design and Automation Association.