Detecting Data Store Access Conflict in Simulink by Solving Boolean Satisfiability Problems

Zhi Han and Pieter J. Mosterman
MathWorks, Three Apple Hill Drive, Natick, MA, 01760, USA

***Abstract*— This paper presents a method to statically analyze a Simulink® model to detect two potential problems with data store memory blocks: (i) a value may be read from a data store before it is written and (ii) a data store may be overwritten before its value is read by other blocks. The analysis employs a Boolean satisfiability (SAT) solver and so obviates extensive testing by means of simulation. It is illustrated how this supports model elaboration in Model-Based Design by performing the analysis on a task model of a digital controller implementation.**

I. Introduction

The use of digital control has become the norm in modern engineered systems. In particular, complex embedded systems such as the Joint Strike Fighter rely on software consisting of millions of lines of code to enable operation [25]. Taking a multitude of control laws from their mathematical representation to an integrated computational implementation has thus become a critical challenge in the design of such embedded control systems. This calls for a gradual refinement of specifications so as to:

- allow quick iterations over incremental design choices and their validity and feasibility, and
- minimize the distance to implementation and the chance of introducing errors.

Also, capturing specifications at different levels of detail with a comprehensive language allows easy communication between all stakeholders and potential issues can be traced back to and solved at different points in the design [15].

In general, the myriad implementation aspects are preferrably addressed incrementally where the order depends on preference, processes, and workflows. The execution aspects of a control law implementation typically involve first deriving a discrete-time representation of the initial continuous-time control law. This representation is then converted into a task-based representation that can be executed by the operating system running on digital hardware. Further implementation considerations are, for example, the use of arguments in functions or using globals for the exchange of data, the use of fixed-point vs. floating-point data types, and partitioning into (potentially reusable) functions.

Model-Based Design revolves around the use of a computational representation of specifications; the *models* of the system under design [15], [22]. The computational form allows automation of many of the required design activities such as computing dynamic behavior, implementating a specification in software, and generating tests. Combined with a sufficiently rich modeling formalism, Model-Based Design provides critical competitive product development advantages (*e.g.*, [16]).

An important design consideration is the task structure and schedule of the control law computations. In case values are communicated between tasks through shared memory, the schedule has to be carefully designed to prevent inadvertent use of stale values. In general, this can prove to be a challenging assignment, especially with schedules that allow conditional task execution. The potential data store access problems, such as reading a value before it was written quickly becomes intractable to the designer, especially if blocks have to be explored across the subsystem hierarchy.

This work employs a Boolean satisfiability (SAT) solver (*e.g.*, [21], [24]) to analyze data store access patterns directly on a Simulink® [19] block diagram representation so as to facilitate incremental design of implementation effects. The static nature of the analysis obviates exhaustive testing (*e.g.*, by simulation) to verify that a given memory access pattern does not arise in any execution. The SAT solver input consists of the block execution relationship and the possible run-time problem encoded as a Boolean formula. This formula is true if and only if the specified pattern of memory access contention can occur and the *counterexample* is displayed to the user in an execution graph. Otherwise, no execution can evidence the offending pattern.

Section II first discusses and compares some related work. Section III motivates the problem based on complications in implementing an adaptive controller. Section IV explains block execution in a Simulink model. Section V introduces the data structure and main procedure for memory access checking. Section VI presents the encoding of the execution structure as Boolean formulae. Section VII illustrates the method first with an abstract example and then the adaptive controller. Section VIII evaluates and concludes this work.

II. Related work

While dynamic analysis by numerical simulations is well established in industrial Model-Based Design, static analysis is rapidly gaining popularity. A number of software tools have been developed to use formal methods for static analysis of Simulink models (*e.g.*, [7], [9], [12], [14], [20]).

Most of these tools require translating the Simulink models to a representation amenable to formal analysis (*e.g.*, [1], [5], [8], [26]). The analysis may not require simulation of the original model but usually has much greater computational complexity. Because Simulink models are mostly hybrid dynamic systems, scalability makes it difficult to

apply formal methods. The translated Simulink models often extend in size beyond the capability of most of the available formal verification tools (because of the inefficiency in the translation algorithm or the complexity of the original models), requiring the models to be significantly simplified [13]. Formal methods may be combined with dynamic simulations to extend their scope and handle relatively large Simulink models (*e.g.*, [2], [18]).

Using existing general purpose formal method tools, a straightforward approach would specify the data store access problem as a *property* to verify. However, such an approach suffers from the limited capability of the formal method tools. Based on the model structure, this work develops a dedicated method that can be simpler and more efficient because it does not require considering all dynamics in the model, as is the case with existing formal method tools. Because the data store access problem only concerns the execution timing of individual blocks, the dynamic properties of the model are unrelated and abstracted away in the analysis. Analysis results are almost instantaneous even when applying the method to sizable Simulink models (*e.g.*, models with thousands of blocks). Moreover, because only the model structure is examined, block-specific data is not needed and partially implemented models can be checked. This supports finding problems early in the design. Note that because the model dynamics are abstracted away, this method can be conservative in that it may report problems on models that are not problematic for certain input data and/or initial state.

Recent algorithmic advances have enabled SAT solving of problems with tens of thousands of variables [24], [21], [11]. SAT solvers are popular in the application of formal methods [10], [3] and because the method presented here is specialized to analyze model structure, the SAT problems are usually quite small (a few hundred of Boolean variables). Future work will investigate if the SAT problem could include more information of the model dynamics to exploit the power of SAT solvers and make the analysis less conservative.

Beyond Simulink models, similar data synchronization issues have been discussed in other contexts (*e.g.*, [6], [12], [17]). Depending on the modeling environment, these issues may be resolved by enforcing certain execution semantics or employ a similar detection diagnostic [17].

III. An Adaptive Controller

To illustrate, an adaptive controller for a first-order plant model is designed and prepared for digital implementation.

A. Adaptive Control Design

In the model reference adaptive control structure in Fig. 1 [4] a plant model of the form

$$\frac{dy}{dt} = -ay + bu \tag{1}$$

is used, with y the plant output to be controlled, u the plant input, and parameters a and b. The reference model defines the trajectory for the plant output to follow

$$\frac{dy_m}{dt} = -a_m y_m + b_m u_c, \tag{2}$$

with y_m the reference plant output, u_c the commanded plant output, and parameters a_m and b_m. The adaptive control is implemented as an algebraic equation

$$u(t) = \theta_1 u_c(t) - \theta_2 y(t), \tag{3}$$

with θ_1 and θ_2 the adaptive parameters. In the s domain, this yields the system output

$$y = \frac{b\theta_1}{s + a + b\theta_2} u_c. \tag{4}$$

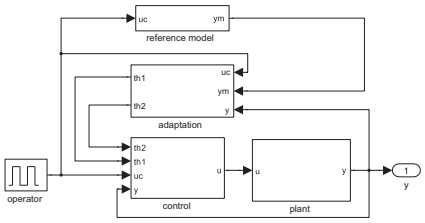


Fig. 1. Model reference adaptive control

A cost function $J(\theta_1, \theta_2) = \frac{1}{2}e^2$ of the error $e = y - y_m$ defines the controller performance. The parameters $\theta = [\theta_1 \ \theta_2]^T$ are adapted over time to minimize J according to

$$\frac{d\theta}{dt} = -\gamma \frac{\partial J}{\partial \theta} = -\gamma \frac{\partial J}{\partial e}\frac{\partial e}{\partial \theta} = -\gamma e \frac{\partial e}{\partial \theta}, \tag{5}$$

with γ a convergence constant. This adaptation can be computed by taking the partial derivative

$$\begin{array}{l} \frac{\partial e}{\partial \theta_1} = \frac{b}{s+a+b\theta_2} u_c \\ \frac{\partial e}{\partial \theta_2} = \frac{b^2\theta_1}{(s+a+b\theta_2)^2} u_c = \frac{b}{s+a+b\theta_2} y \end{array}. \tag{6}$$

Now, by employing the approximation $s + a + b\theta_2 \approx s + a_m$, the parameter adaptation in Eq. (5) becomes

$$\begin{array}{l} \frac{d\theta_1}{dt} = -\gamma \left(\ \frac{a_m}{s+a_m} u_c \ \right) e \\ \frac{d\theta_2}{dt} = \gamma \left(\ \frac{a_m}{s+a_m} y \ \right) e \end{array} \tag{7}$$

and the ideal parameter values after convergence are then $\theta_1 = \theta_1^0 = \frac{b_m}{b}$ and $\theta_2 = \theta_2^0 = \frac{a_m - a}{b}$.

B. Towards a Digital Implementation

To prepare the controller design for automatic code generation a number of implementation choices have to be made:

- The sample time of the controller is chosen to be $15\ ms$. The corresponding *zero-order hold* discrete approximation is derived.
- A task-based representation is derived by converting the discrete transfer functions into a sample time independent computational form.
- The communication of data between tasks is chosen to be done through globals in shared memory.
- Effects of the device drivers that connect the controller to operator input and to the plant are modeled.

Figure 2 shows the result. On the left-hand side, the operator is shown as a separate task that writes to a Data Store Memory (DSM) that models the device memory *DSM_devc*. On the right-hand side the plant is shown as a separate process that operates in continuous time. The digital to analog converter (DAC) and the analog to digital (ADC) converter

operate on device memory *DSM_devi* and *DSM_devo*, respectively. In the center, the controller is shown. It comprises eight tasks: (i) read from the operator input device, (ii) write to the DAC, (iii) read from the ADC, (iv) compute the reference model output, (v) compute the controller output, (vi) update the parameters, (vii) store instrumented variables for monitoring, and (viii) handle possible exceptions. These tasks store the reference model output in *DSM_ym* and the control parameters in *DSM_th1* and *DSM_th2*.

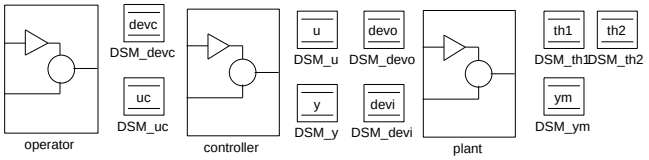
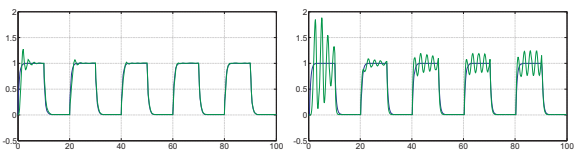


Fig. 2. Implementing the ideal controller

Because a DSM block represents memory that can be shared between different parts of a Simulink model the need to explicitly schedule the separate tasks becomes apparent. Data Store Read (DSR) and Data Store Write (DSW) blocks access the value of a DSM block. A DSW block that executes writes its input to the corresponding DSM. A DSR block that executes reads the content of the corresponding DSM block and puts the value on the DSR output port.

The semantics of the declarative part of a Simulink block diagram is specified by its fixed-point behavior, akin to fixed-point semantics in a general denotational framework such as *lambda calculus* [23]. To efficiently compute the fixed-point values at each time step, this schedule of computations is automatically generated based on the input/output relations between blocks. This schedule determines the memory read and write moments. Because there is no explicit input/output relation associated with the read/write order of DSR and DSW blocks, delays may be introduced (*i.e.*, a computed value is not accessed until a next time step). Likewise, the software design engineer may choose a schedule where the control task reads from the parameter memory before it has been updated, thus introducing a delay as well.

Figure 3 illustrates the effect of such inadvertent scheduling by comparing behavior of the ideal controller using continuous-time differential equations in Fig. 3(a) with the tasked controller in Fig. 3(b). Analysis of the memory access behavior for each DSM is necessary to identify the culprit of the undesired (persistent oscillatory) behavior. This reveals whether a DSR block attempts to access its DSM before the DSW block has written to it or whether more than one DSW block run in the same simulation step.



(a) Continuous-time (b) A tasked execution

Fig. 3. Adaptive control comparison

For scheduling algorithms that rely on complicated conditional task execution, this analysis quickly becomes intractable to the design engineer. While simulation can be employed to detect when a memory read occurs before a write, this result only holds for the one particular execution. Consequently, a static analysis of all potential memory access contention is preferred.

IV. Block Execution in a Simulink® Model

Simulink determines an execution order that requires evaluating each block only once for computing the fixed-point solution at a time step. This work assumes no cycles exist and so the *sorted order* that is derived is based on how the block inputs and outputs are connected. All blocks that compute as output the input to a given block are evaluated before the given block is evaluated and are called *dependent* blocks of the given block. In a simulation step, a block can then only start execution after all its dependent blocks have executed.

A *subsystem* contains blocks and other subsystems to define its behavior and support hierarchical decomposition. A subsystem is called *virtual* if it only serves a graphical structuring purpose without having a semantic bearing. When the sorted order is derived, a virtual subsystem is replaced by its content in a 'flattening' stage.

Nonvirtual subsystems are elements in the sorted order that have an internal sorted order and so create an execution hierarchy. Here, nonvirtual systems are considered to be of block type. A nonvirtual subsystem is either *atomic* or *conditional*. Similar to a block, an *Atomic Subsystem* commences execution when all its input has been computed. It then executes all its internal blocks and completes execution when all these blocks have completed execution. Blocks that take the output of an atomic subsystem as input can only start executing after the subsystem has completed execution.

Conditional subsystems can be classified based on how their execution is predicated on their control signals [19]:

- An *Enabled Subsystem* executes at the time step when its control signal becomes high. It remains active as long as its control signal is high.
- A *Triggered Subsystem* executes at the time step when its control signal exhibits a rising or falling edge. It does not remain active.
- A *Control-flow Subsystem* executes its content expressed by control-flow statements (*e.g.*, *if-then*, *while*, and *do*) as an atomic subsystem. It does not remain active.
- A *Function-call Subsystem* executes its content as an atomic subsystem. It does not remain active.

Here, blocks that generate the control signal are called the *control* blocks of the conditional subsystems. Enabled Subsystems and Triggered Subsystems are included in the sorted list and execute accordingly. If disabled by their control signal, the next block on the sorted list is executed. Control-flow Subsystems and Funtion-call Subsystems execute immediately when their *control block* initiates execution. As such, this implements an *imperative* semantics mixed with the *declarative* semantics of other blocks and subsystems.

V. Main results

The *execution graph* captures all dependencies that determine whether a block can execute at some point in a

simulation step: its dependent blocks, its parent subsystems, and the control blocks of its parent subsystems.

Definition 1 (Execution graph): The execution graph of a model M is a directed graph $\mathcal{G} = (V, E)$, where

- V is a set of nodes, such that:
  - A unique node $v_0 \in V$ represents the model.
  - For any $v \in V$, $v \neq v_0$ there is a unique corresponding nonvirtual block in M.
- $E \subseteq V \times V$ is a set of edges and $v_1 \rightarrow v_2$ is written iff $(v_1, v_2) \in E$, which can be any of the classes:
  - v_2 corresponds to a conditional subsystem block and v_1 to the control block (control relationship).
  - The block corresponding to v_2 depends algebraically on the block corresponding to v_1 (data dependency relationship).
  - v_1 corresponds to a subsystem block and v_2 to a block inside the subsystem block (parent-child relationship).

Note that it is not required that each block in the model has a corresponding node in $\mathcal{G}$ because typically only a subset of blocks is of interests when considering the execution ordering of DSW and DSR blocks.

Input: A Simulink model M, a data store memory D
Output: A Resulting execution graph $\mathcal{G}$
Procedure:
COMPILE_MODEL(M)
$\{B_i, i = 1, \cdots, k\}$ = FIND_BLOCKS(D)
$\mathcal{G}$ = CREATE_GRAPH($M, \{B_i\}$)
FOREACH pair of datastore blocks (B_{p1}, B_{p2}) that may have data access issues
    ϕ = CREATE_SAT_FORMULA($M, \{B_{p1}, B_{p2}\}$)
    [sat, $assignments$] = SAT_SOLVE(ϕ)
    IF sat
        HIGHLIGHT_GRAPH($\mathcal{G}$, $assignments$)
    ELSE
        Report a negative result
    ENDIF
ENDFOREACH

Fig. 4. The main procedure **ANALYZE_DATASTORE**

Figure 4 shows the algorithm to statically check a Simulink model for Data Store Memory access problems. First, DSR and DSW blocks are identified in the model. Next, an execution graph of the model is created as well as a Boolean formula by traversing the execution graph and corresponding blocks in the model. The formula encodes the execution logic of the model and the Data Store Memory access pattern to be detected. A SAT solver then solves the Boolean formula. If negative, the problem cannot occur in the model. If positive, the problem can occur. An assignment of the Boolean variables that satisfy the formula is returned and used to create an annotated execution graph to display a snapshot of the offending execution.

Figure 5 shows the pseudo code to create the execution graph for the analysis. A breadth-first search starts from the DSW and DSR blocks to visits all relevant blocks in the model necessary to construct the execution graph.

When a block is visited, the model structure is examined to find the set of blocks that must be executed before the current block can start execution. This includes the dependent blocks, the parent subsystem block, and the control

Input: A Simulink model M and a set of blocks B_i associated with a data store, $i = 1, \cdots k$.
Output: Execution graph $\mathcal{G}$
Procedure:
INITIALIZE_QUEUE
ENQUEUE(B_i) $i = 1, \cdots, k$
WHILE QUEUE_NOT_EMPTY
    B = DEQUEUE
    IF B is a conditional subsystem
        Find control blocks C_i of B, $i = 1, \cdots, m$
        FOREACH C_i
            Add nodes and connections for C_i
            ENQUEUE(C_i) if C_i has not been visited
        END FOREACH
    END IF
    Find blocks D_i that B depends on, $i = 1, \cdots, n$
    FOREACH D_i
        Add nodes and connections for D_i
        ENQUEUE(D_i)
    END FOREACH
    Find the parent system P of B
    Add nodes and connections for P
    IF P is not the root model ENQUEUE(P)
END WHILE

Fig. 5. The **CREATE_GRAPH** procedure to generate an execution graph

blocks if the block is a conditionally executed subsystem. The execution graph then includes these blocks and the corresponding edges. If a block is not executed prior to the current block (*e.g.*, a block in the same subsystem that is not in the transitive closure of the input/output graph from the current block), the block is not visited and not further considered because it does not play a role in determining the execution of the set of DSR and DSW blocks.

If there are multiple DSM blocks present in the model, one execution control graph is created for each pair of DSR/DSW blocks. All of the analysis is performed individually for the DSW and DSR blocks associated with each DSM block.

VI. Boolean Satisifability Formulae

The Boolean formulae to be checked specify the relationship of the execution status of the blocks and subsystem blocks corresponding to the execution graph in the simulation step.

A. Elements of the Encoding

Consider the execution graph given as $\mathcal{G} = (V, E)$. For each node $v \in V$, two variables v_S and v_E, are introduced where Table I lists the truth table of the variables.

TABLE I
Boolean variables for block execution

v_S	v_E	**Block (or subsystem) status**
F	F	Block execution has not started.
T	F	Block execution has started but not completed.
T	T	Block execution is completed.
F	T	Block is disabled.

For an edge $e \in E$ that corresponds to a signal in the model, a Boolean variable e_R is introduced if the edge corresponds to a data dependency, where e_R is true iff the signal corresponding to e is already output by the block generating the signal. If the edge corresponds to a control signal, two Boolean variables e_R and e_A are introduced for the edge, where e_A is true iff both e_R is true and the control block sends an activation signal (*e.g.*, function-call, action, or trigger signal), see Table II.

TABLE II

BOOLEAN VARIABLES FOR CONTROL SIGNALS

e_R	e_A	**Control signal status**
F	F	Control signal not ready.
T	F	Control signal is deactivate signal.
T	T	Control signal is activate signal.
F	T	Not possible

B. Encoding the Simulink® Execution Logic

For each node $v \in V$ that corresponds to a block or an atomic subsystem (*i.e.*, nonvirtual subsystem with no control ports) the following set of Boolean formulae is created to specify execution conditions:

- A block can only execute if its dependent data inputs have completed. For any edge $e = (u, v) \in E$ for some u, and e that corresponds to a data signal, $v_S \rightarrow e_R$.
- A block can only start executing if its parent system has started execution. For any edge $(p, v) \in E$ that is a parent-child edge, $v_S \rightarrow p_S$.
- A parent system can only complete its execution if its children block execution has completed. For any edge $(p, v) \in E$ that is a parent-child edge, $(p_E \wedge p_S) \rightarrow (v_E \wedge v_S)$.
- A block is disabled if and only if its parent is disabled. For any edge $(p, v) \in E$ that is a parent-child edge, $(\neg v_S \wedge v_E) \leftrightarrow (\neg p_S \wedge p_E)$.
- The output of a block is ready when it completes execution. For any edge $o = (v, u) \in E$ for some u, and the edge o that corresponds to a data signal, $o_R \leftrightarrow v_E$.

For a control block, the Boolean formulae for blocks are generated as well as the following formulae for the control:

- For control blocks that implement '*If-else*' or '*Switch-case*' logic, at most one of the output control signals is active, that is, for a pair of edges $e_1 = (v, v_1)$ and $e_2 = (v, v_2)$ where v correspond to a block implementing '*If-else*' or '*Switch-case*' logic, $\neg e_{1A} \vee \neg e_{2A}$.
- Each control signal can only be active when the signal is ready. For an edge e that corresponds to a control signal, $e_A \rightarrow e_R$.

For other control blocks (*e.g.*, Function-call Generator blocks), no special logic is specified and hence the control block may output any control signal.

The root model always executes, formulated as v_0. The following formula are further generated for the root model and subsystems:

- At any point in time, only one of the blocks in the subsystem is actively executing. For each pair u, w such that $(v, u) \in E$ and $(v, w) \in E$ that both correspond to parent-child relationship, $\neg w_S \vee \neg u_S \vee w_E \vee u_E$.

For each conditional subsystem block in the graph, the formulae for regular blocks are generated as well as the following Boolean formulae:

- Function-call and Action Subsystems execute if the input signal is active. For an edge $e = (c, v)$ that corresponds to an incoming control signal, $e_A \leftrightarrow v_S$. The subsystem is disabled if the parent is disabled or the control signal is inactive. Suppose p corresponds to the parent system, then $(\neg v_S \wedge v_E) \rightarrow ((\neg p_S \wedge p_E) \vee \neg e_A)$.
- Enabled and Triggered Subsystems may execute when the input signal is active. For an edge $e = (c, v)$ that corresponds to an incoming enable or trigger signal, $v_S \rightarrow e_A$. The subsystem is disabled if the control signal is inactive or the parent system is disabled. Suppose p corresponds to the parent system, then $(\neg v_S \wedge v_E) \rightarrow ((p_S \wedge p_E) \vee \neg e_A)$. The subsystem must have completed execution if the parent system is not disabled, has completed execution, and the incoming control signal is an activate signal, $(p_S \wedge p_E \wedge e_A) \rightarrow (v_S \wedge v_E)$.

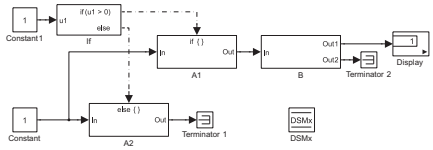
The DSM access problems are encoded as follows:

- For a pair of DSR and DSW blocks that correspond to the same DSM block, it is checked whether the DSR block can start its execution while the DSW block has not executed. Suppose $dsr \in V$ corresponds to the DSR block and $dsw \in V$ corresponds to the DSW block, the satisfiability of the following formula $dsr_S \wedge dsr_E \wedge \neg dsw_S$ must be solved.
- For a pair of DSW blocks that correspond to one DSM block, it is checked whether the DSW blocks can execute in the same simulation step. Suppose $dsw_1 \in V$ and $dsw_2 \in V$ correspond to the DSW blocks, the satisfiability of the following formula $dsw_{1S} \wedge dsw_{1E} \wedge dsw_{2S} \wedge dsw_{2E}$ must be solved.

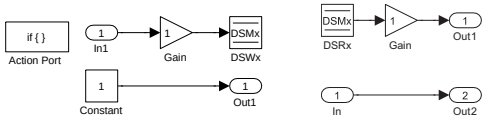
The conjunction of all the Boolean formulae is satisfiable if all of the sub-formulae are satisfiable and an execution of the model allows the access pattern.

VII. CASE STUDIES

Figure 6 shows the block diagrams of a Simulink model with If-action Subsystems *A1* and *A2* that are conditionally executed by the *If* block. The model has a DSM block *DSMx* in the root-level system. There is a DSW block *DSWx* inside subsystems *A1* and inside subsystem *A2*, see Fig. 6(b). There is a DSR block *DSRx* inside subsystem *B*, see Fig. 6(c). Subsystem *B* reads the value held by *DSMx*, computed either by *A1* in the if branch or *A2* in the else branch. The SAT formula for the block *DSRx* to execute before *DSWx* is satisfiable and the assignment Boolean values to the logical variables is used to highlight the execution graph in Fig. 7.



(a) System model



(b) Subsystems *A1* and *A2* (c) Subsystem *B*

Fig. 6. A Simulink® model with If-action Subsystems

The counterexample in Fig. 7 can be interpreted in the model in Fig. 6. The control block *If* deactivates *A1* and activates *A2*, therefore the DSW block *DSWx* in subsystem *A1*

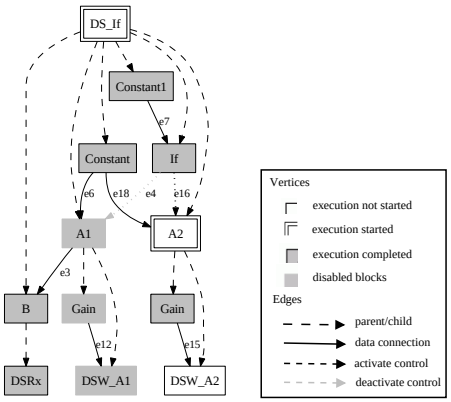
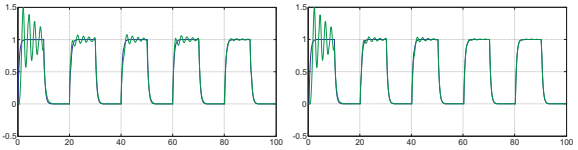


Fig. 7. The execution graph of the model

does not execute. When the DSR block *DSRx* in subsystem *B* executes, the block is reading data from an uninitialized data store as subsystem *A2* may not have executed yet.

Analysis of the adaptive controller in Section III generates an execution graph for each of the nine DSM blocks in the model in Fig. 2. The method identifies the culprit for the anomalous behavior in Fig. 3(b) to be the potential read before write of DSM block *DSM_th1*. By performing the parameter adaption before computing the control signal, this Data Store Memory access problem is eliminated. Figure 8 shows how the corrected tasked controller performance in Fig. 8(b) compares with a time-discretized approximation in Fig. 8(a) of the original controller in Fig. 3(a). Note that the considerable oscillatory behavior is an exaggeration for illustration purposes that is achieved by using a high covergence factor ($\gamma = 9.8$) in all models presented in this paper. In practice, the problem manifestation may be much more subtle and difficult to detect based on simulation.



(a) Discrete time (b) A tasked execution

Fig. 8. Discrete-time and proper tasked execution

VIII. Conclusion and Evaluation

A method to statically analyze shared memory access at the model level was presented. The method supports model elaboration of Model-Based Design by facilitating early evaluation of a control law implementation. The method was applied to an industry benchmark model with 1832 Simulink blocks, 112 Data Store Memory blocks, 144 Data Store Read blocks, and 82 Data Store Write blocks. The smallest problem has an execution graph with 24 nodes and 48 edges. The corresponding SAT problem has 144 variables. The largest problem has an execution graph of 124 nodes and 235 edges. The corresponding SAT problem has 718 variables. These problems are quite small for modern SAT solvers.

Acknowledgements

The authors thank Alongkrit Chutinan, Murali Yeddanapudi and Pripal Dang for their help and support.

References

[1] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *Electr. Notes Theor. Comput. Sci*, vol. 109, pp. 43–56, 2004.

[2] R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *International Conference On Embedded Software*, pages 89–98, 2008.

[3] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. Mcmillan. An analysis of SAT-based model checking techniques in an industrial environment. pages 254–268. 2005.

[4] K. J. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley, Reading, Massachusetts, 2 edition, 1995.

[5] M. B, A. Bhatnagar, and S. Roy. Automatic translation of Simulink models into input language of a model checker. In *ICFEM*, pages 606–620, 2006.

[6] P. Caspi, J.-L. Colaço, L. Gérard, M. Pouzet, and P. Raymond. Synchronous objects with scheduling policies: introducing safe shared memory in Lustre. In *ACM SIGPLAN/SIGBED conference on LCTES*, pages 11–20, 2009.

[7] A. Chapoutot and M. Martel. Abstract Simulation: a Static Analysis of Simulink Models. In *International Conference on Embedded Systems and Software*, pages 83–92, 2009.

[8] C. Chen, J. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.

[9] A. Chutinan and B. H. Krogh. Compuational techniques for hybrid system verification. *IEEE Transaction on Automatic Control*, 48(1):64–75, Jan 2003.

[10] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, pages 7–34, 2001.

[11] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[12] Esterel Technologies. *SCADE suite*. http://www.esterel-technologies.com/products/scade-suite/.

[13] A. Fehnker and B. H. Krogh. Hybrid system verification is not a sinecure - the electronic throttle control case study. In *ATVA*, pages 263–277, 2004.

[14] R. E. L. Harmmon and C. Hôte. Automatic engine control code generation with integrated automatic static code verification. Technical report, Cummings Engines, Inc. and PolySpace Technologies, 2006.

[15] J. W. Jacobs. *Model-based Application Development for Massively Parallel Embedded Systems*. PhD dissertation, University of Twente, Enschede, Netherlands, 2008. ISBN: 978-90-365-2752-1.

[16] H. Jones. Return on investment in Simulink for electronic system design. Technical report, International Business Strategies, 2005.

[17] E. A. Lee and T. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[18] F. Lerda, J. Kapinski, H. Maka, E. M. Clarke, and B. H. Krogh. Model checking in-the-loop: Finding counterexamples by systematic simulation. In *American Control Conference*, pages 2734–2740, 2008.

[19] MathWorks. *Simulink® 7: Using Simulink*, March 2010.

[20] MathWorks. *Simulink® Design Verifier™: User's guide*, March 2010.

[21] M. W. Moskewicz and C. F. Madigan. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[22] P. J. Mosterman, S. Prabhu, and T. Erkkinen. An industrial embedded control system design process. In *Proceedings of The Inaugural CDEN Design Conference*, pages 02B6–1 – 02B6–11, Montreal, July 2004.

[23] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, Hoboken, NJ, 1992.

[24] J. L. M. Silva. GRASP: A new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[25] M. Sullivan. JOINT STRIKE FIGHTER–progress made and challenges remain. Technical Report GAO-07-360, United States Government Accountability Office, March 2007.

[26] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):779–818, 2005.