Electronic Communications of the EASST
Volume 46 (2011)

Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)

Combining Model Checking and Discrete-Event Supervisor Synthesis

Nicolas Chausse, Helen Xu, Juergen Dingel and Karen Rudie

15 pages

Guest Editors: Jens Bendisposto, Cliff Jones, Michael Leuschel, Alexander Romanovsky
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Combining Model Checking and Discrete-Event Supervisor Synthesis

Nicolas Chausse[1], Helen Xu[1], Juergen Dingel[1] and Karen Rudie[2]

[1] chausse, helen, dingel@cs.queensu.ca, School of Computing, Queen's University, Canada

[2] karen.rudie@queensu.ca, Dept. of Elec. and Comp. Eng., Queen's University, Canada

Abstract:

We present an approach to facilitate the design of provably correct concurrent systems by recasting recent work that uses discrete-event supervisor synthesis to automatically generate concurrency control code in Promela and combine it with model checking in Spin. This approach consists of the possibly repeated execution of three steps: manual preparation, automatic synthesis, and semi-automatic analysis. Given a concurrent Promela program C devoid of any concurrency control and an informal specification E_{in}, the preparation step is assumed to yield a formal specification E of the allowed system behaviours and two versions of C: C_e to identify the specification-relevant events in C and enable supervisor synthesis, and $C_{e,a}$ to introduce "checkable redundancy" and used during the analysis step to locate bugs in: the specification formalization E, the event markup in C_e, or the implementation of the synthesis. The result is supervised Promela code C_{sup} that is more likely to be correct with respect to E *and* E_{in}. The approach is illustrated with an example. A prototype tool implementing the approach is described.

Keywords: Concurrency control, formal verification, control theory, discrete-event systems, controller and supervisor synthesis.

1 Introduction

The poor integration between computer science and electrical engineering in academia has been observed before. In [HS07], Henzinger and Sifakis blame the "wall" between these two disciplines for keeping the "potential of embedded systems" at bay. Indeed, the potential for fruitful interaction between them seems large. Consider, for instance, Discrete-Event Systems (DES) control theory, a branch of control theory which is concerned with the *Supervisory Control Problem* (SCP), i.e., the automatic synthesis of a supervisor (controller) S that restricts the execution of an unrestricted discrete-event system G (called "plant") to enforce some specification E. DES theory originated in the 1980s [RW87, RW89] and offers a large body of research on the SCP which, for instance, considers different formalisms to represent S, G and E including finite state automata (FSA), Petri nets, and the mu-calculus [CL08, ZS05]. Recent work has shown how results and tools from DES theory can be used to alleviate the challenges of concurrent programming. In [WLK+09, WCL+10], automatically generated supervisors are used to guarantee deadlock-free execution of multi-threaded code, based on a structural analysis of a Petri-net representation of the plant. In [DDR08], standard DES based on FSAs is employed to generate

supervisors that enforce deadlock-freedom and safety properties (also expressed as FSAs) on Java programs with static concurrency. In [ADR09], this work is extended to dynamic concurrency which then requires the use of Petri nets.

We extend this line of work and suggest the integration of DES theory with model checking by combining the constructive and generative aspects of DES theory with the analysis and bug detection capabilities of Spin. We aim to facilitate the development of provably correct concurrent systems by increasing the degree of automation. This paper makes the following contributions: (1) The work in [DDR08] is recast in Promela. Given an unrestricted system C expressed in Promela and a specification E expressed as a FSA, a supervised system C_{sup} is automatically generated and is guaranteed to satisfy E and deadlock-freedom. Moreover, the supervisor component in C_{sup} is provably minimally restrictive (maximally permissive), i.e., any behaviour in C but not in C_{sup} will violate E or deadlock-freedom. (2) Despite the theoretical guarantees, bugs can still creep in not only in the various synthesis steps' implementation, but also in the inputs to the synthesis steps, all of which are, at least partially, manually created. We show how model checking can be used to debug them. (3) We describe a prototype tool using Spin and show how Spin's support for shared-memory and message-passing concurrency can be leveraged to generate supervisors supporting the two concurrency paradigms and to optimize the analysis of the combined system. A detailed example illustrates the approach and the tool's utilization.

This paper is structured as follows: Related work is reviewed in Section 2 and relevant background on DES theory is given in Section 3. Section 4 describes our approach and Section 5 illustrates it with an example. Section 6 describes our prototype tools and Section 7 concludes.

2 Related Work

Automatically generating parts of concurrent systems from specifications has been an active research topic. We focus here on approaches that combine synthesis and formal analysis via model checking. While the use of DES in software development and execution has been suggested before [RW90, RW92a, Laf88, TMH97, WKL07], generating control code for concurrent software has received particular interest recently. The work of two authors of this paper on using DES for generating concurrency control code has already been mentioned [DDR08] where the JPF model checker was used to validate the generated supervisor code, but not the manually created inputs. Moreover, despite recent advances in software model checking, model-level analyses are still more likely to be tractable rather than at code-level. Independently, Wang *et al.* have used DES to obtain supervisors that guarantee deadlock-freedom [WLK+09, WCL+10] where concurrent programs are represented as Petri nets and deadlock freedom is characterized by the absence of reachable empty siphons. Our work in this paper (and [DDR08]) is based on FSAs and supports general safety properties rather than just deadlock-freedom. Also, no support for analysis of the generated artifacts is mentioned in [WLK+09, WCL+10]. Timed DES is based on timed automata; recently, UPPAAL-TIGA has been used for an industrial case study involving climate control systems [BCD+07] where the synthesis and analysis capabilities of UPPAAL-TIGA have been combined with Simulink and Real-TimeWorkshop to provide a complete tool chain for synthesis, simulation, analysis and automatic generation of production code. The work in [GPT06] uses symbolic model checking for supervisor synthesis from specifications given

in CTL specifications and a plant description given in NuSMV. The work in [ZS05] introduces DES theory based on the mu-calculus and thus generalizes Ramadge and Wonham's standard DES theory. However, no tool supporting the generalization appears to be available.

There exists additional work that does not make explicit use of DES theory. For instance, some work is aimed at facilitating software architecture component composition (e.g., [TI08, BBC05]). In [TI08], Tivoli and Inverardi generate coordinators which enforce a given global coordination policy [TI08] where components are assumed to adhere to a coordinator-based architectural style and message sequence charts are used for behavioural interface specification. Correctness and maximal permissiveness (called completeness) are proved and the work has been integrated with CHARMY, a tool for architectural analysis. Despite many differences in technical details and terminology, the approach is similar to supervisor synthesis[1]. In the context of concurrent programming, the approach presented by Deng *et al.* explicitly shares our interest in supporting the combined use of synthesis and verification [DDHM02]. It generates synchronization statements for concurrent Java code from invariant specifications and the new code can be fed into the Bandera model checker for analysis. Some related work appears in the literature as environment (assumption) generation. For instance, in [GPB05], the LTSA tool is used to determine the weakest assumptions that the concurrent environment E of a component C has to satisfy such that the composition of C and E satisfies some specification B where E, C, and B are given as FSAs. LSTA also supports model checking. Synthesis has also been used to achieve fault-tolerance. In [AAE04], a method is presented for the synthesis of fault-tolerant concurrent programs from specifications expressed in the temporal logic CTL. However, no implementation allowing the integration with CTL model checkers such as nuSMV is mentioned. Finally, in [IST07] and [IS08], CSP||B is used to control machines or processes via control "annotations" which may represent states, next operations or control flow. A synthesis process is used to: verify the annotations against the machine, manually produce a "Controller" and verify it against the annotations, and finally refine if needed.

We conclude that while the integrated use of synthesis and formal verification has been suggested before, our work differs from each of the existing approaches in at least one of the following two aspects: it uses Spin, one of the most popular and powerful model checkers available; it explicitly uses DES theory and thus allows the large body of existing results and tools to be leveraged. Interestingly, the recent interest in autonomic and adaptive software has produced proposals to design software directly informed by control theory [MPS08, Dah10]. However, so far, controller synthesis does not appear to be part of this research agenda. In [Dah10], validation and verification of autonomic and adaptive systems are singled out as particularly important research topics.

[1] In [TI08, p. 206], it is claimed that supervisor synthesis based on DES requires explicit specification of the deadlocking behaviours; this, however, is not the case.

3 Background

3.1 DES Theory

In DES theory, systems are modelled by FSAs called *plants*. Transitions represent events that are either *controllable* (can be enabled or disabled at will) or *uncontrollable* (may happen arbitrarily). In a *non-blocking* model, all states are *reachable* (from the initial state) and *co-reachable* (lead to a final state) which implies the absence of deadlocks and livelocks. A specification describing a plant's desired behaviour can be modelled using specification FSAs and is called the *specification*, or *legal language*. A specification E *is controllable* with respect to plant G if for any series s of events in G and legal in E (s is in E's prefix closure), there is no uncontrollable event σ that can then happen in G and that is illegal in E ($s\sigma$ is not in E's prefix closure).

Given a specification E and plant G, where E is not necessarily controllable with respect to G, we want to get the least restrictive sub-specification (or largest sub-language) $K \subseteq E$ such that K is controllable with respect to G. If there is no such nonempty subset of E then $K = \emptyset$. If E is controllable with respect to G, then $K = E$. We call a *recognizer* S for K the *supervisor* or the *supremal controllable sub-language* of E with respect to G, denoted $sup\underline{C}(G,E)$ [CL08]. The supervisor is also modelled with an FSA and will *control* G by *enabling* and *disabling* G's controllable events. When a plant G is controlled by a supervisor S, the resulting behaviour is given by the intersection of the language accepted by G and the language accepted by S and is captured by a FSA denoted as S/G.

Composing Specifications and Processes: The plant G and the specification E may consist of several parallel processes G_i and sub-specifications E_j, respectively. We assume that the sub-specifications share all events (i.e., use the same set of events), which means that each node in a sub-specification has a self-loop labelled with all the events that do not directly belong to any sub-specification but belong to the processes. Processes, however, may not share all events. We will combine processes and sub-specifications using an operation that forces the FSAs to synchronize on shared (common) events, while allowing independent interleavings of the non-shared events. We will call this operation *synchronous product*[2].

Complexity and Tool Support: The supervisor $sup\underline{C}(G,E)$ can be computed in time $O(n^2m^2e)$ where n and m are, respectively, the number of states in G and E and e is the total number of events in G and E (Section 3.5.3 of [CL08]). The time complexity of the synchronous product operation is $O(mn)$ where n is the number of sub-FSAs provided and m the maximum number of states in all these sub-FSAs. Several DES tools supporting supervisor synthesis are available including IDES [IDE], TCT [TCT], and DESUMA [DES].

3.2 DES Theory for Generation of Concurrency Control Code

As described in Section 2, previous work has already observed that DES theory can be used directly to control the execution of software with respect to certain specifications [DDR08, WLK+09]. The area of application here has been concurrent programming where the supervisor manages concurrent processes such that deadlock-freedom and the safety properties expressed as FSAs are enforced — the generated supervisor inheriting the strong theoretical guarantees

[2] Note that if two FSAs share all events, the synchronous product reduces to language intersection.

offered by DES theory. The key idea is to view the concurrent system as the plant G and to interpret concurrency- or specification-relevant operations in the code as controllable events. To obtain the closed loop system S/G, the event markup in G is replaced by an interaction with the supervisor in which a request by a process in G to execute an operation is only granted by the supervisor if its execution cannot possibly lead to a deadlock or specification violation. The approach requires the (manual or automated) identification of relevant events in the code and then the transformation of the code and the specification into a format supported by current DES tools. For instance, in [WCL+10] concurrent C code is automatically converted into a Petri net by extracting and combining the control flow graph of each of the threads and modelling execution via token flow. In [DDR08], a similar technique is used to convert Java threads into FSAs which are then combined using the synchronous product operation.

4 Combining Supervisor Synthesis and Spin Analysis

A graphical overview of our approach to integrate supervisor synthesis and analysis is given in Figure 1, which shows the flow of artifacts (solid arrows) between possibly nested activities (boxes). Stick figures indicate activities requiring user interaction and the dashed arrow shows control flow.

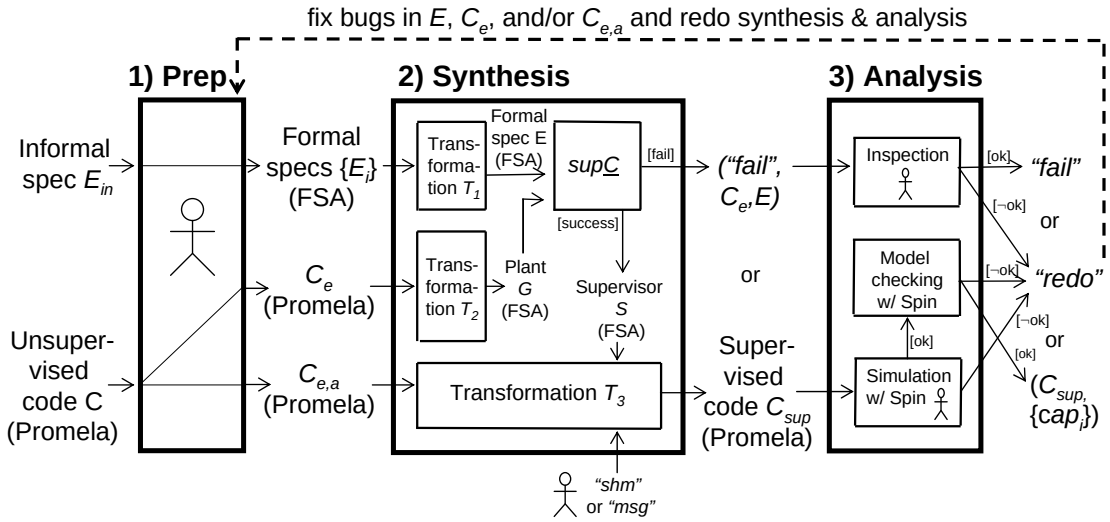


Figure 1: Overview of Approach to Integrate Synthesis and Spin Analysis

This approach recasts the preparation and synthesis steps for concurrency control code generation proposed in [DDR08] using Promela (instead of Java) as the implementation language. Moreover, an additional artifact ($C_{e,a}$) is introduced and the synthesis is followed by an analysis step in which manual inspection, user-guided simulation, and model checking are used to identify bugs in any of the artifacts created during the manual preparation step. If bugs are found, the preparation and the synthesis are redone. We describe each step in more detail.

1) Preparation: The informal specification E_{in} is assumed to express a safety property identify-

ing permissible sequences of events such as precedence constraints, mutual exclusion constraints or capacity constraints. The unsupervised code C is a concurrent Promela program devoid of any concurrency control. The user then (1) translates E_{in} into a collection $\{E_i\}$ of FSAs, (2) marks up specification-relevant events in C to create C_e, and (3) adds assertions and possibly auxiliary variables to C_e to obtain $C_{e,a}$. The transitions in E should distinguish between controllable and uncontrollable events. The assertions in $C_{e,a}$ capture (aspects of) the informal specification E_{in} and offer "checkable redundancy", which will be used in the analysis step to validate E against E_{in}. For instance, a capacity constraint in E_{in} may be checked by an assertion containing a counter variable.

2) Synthesis: Consists of the $sup\underline{C}$ operation, sandwiched between three transformations: T_1 and T_2 to prepare the inputs and T_3 to process the output:

a) The formal specifications E_i are combined into a single one by computing their synchronous product E (transformation T_1 in Figure 1).
b) The unsupervised code with event markup C_e is translated into plant FSA G (transformation T_2). Similar to [DDR08, WCL+10], G is obtained using compiler technology to extract the control-flow graph of every process in C_e and to build FSA-representations. These FSAs are combined by computing their synchronous product.
c) An off-the-shelf DES tool is used to perform the $sup\underline{C}$-operation on E and G.
d) If $sup\underline{C}(G,E) = \emptyset$, the operation fails. Otherwise, the generated supervisor S is automatically implemented in Promela and integrated in $C_{e,a}$ to obtain the supervised code C_{sup} (transformation T_3). Transformation T_3 allows the generation of code that implements the supervision using shared-memory (input *"shm"* in Figure 1) or message-passing (*"msg"*).

3) Analysis: The analysis process is described in Figure 2. If the $sup\underline{C}$-operation fails (line 3),

```
1  input: ('fail',E,Ce) or Csup
2  output: 'fail', 'redo', or Csup
3  if input=='fail' then                                  % SupCon operation failed
4     check that Ce and E are correct wrt C and Ein;      % Manual inspection
5     if bug found then                                   % E and/or event markup in Ce wrong
6        output 'redo' and stop;                          % Fix bug and redo synthesis
7     else output 'fail' and stop                         % Ein may be unenforceable on C; done
8  else
9     simulate Csup in Spin;                              % Does Csup behave as expected? (semi-automatic step)
10    if Csup has unexpected behaviour then
11       output 'redo' and stop;                          % Fix bug and redo synthesis
12    else
13      modelcheck Csup in Spin;                          % Do assertions hold?
14      if violation found then                           % E or assertions in Ce,a must be wrong wrt Ein
15         output 'redo' and stop;                        % Fix bug and redo synthesis
16      else
17         use Spin to determine minimal channel capacities {capi};
18         output (Csup,{capi}) and stop.                 % Done
```

Figure 2: Pseudocode for Analysis Step in Figure 1 (indentation indicates nesting)

it may be because C_e or E are incorrect. For instance, event markup in C_e may be misplaced or missing; E may have incorrect transitions or may erroneously mark a controllable event as uncontrollable. If manual inspection uncovers such an issue (line 4), the preparation and the

synthesis are redone. Otherwise, C_e and E are assumed to be correct (w.r.t. E_{in} and C) and the process ends in a fail (because E is unenforceable on C) (line 7). If the $sup\underline{C}$-operation is successful (i.e., $sup\underline{C}(G,E) \neq \emptyset$), the supervised code C_{sup} is simulated by the user (line 9); if unexpected behaviour is encountered, the preparation and the synthesis are redone; otherwise, C_{sup} is model checked (line 13). Assertion violations indicate that either E or the assertions are incorrect and a new iteration is initiated (line 15). If no violations are found, Spin is used to determine the smallest channel capacities $\{cap_i\}$ necessary to implement C_{sup} and the supervised code C_{sup} is output with $\{cap_i\}$.

4.1 Theoretical Guarantees

Strong guarantees can be given for the result of the $sup\underline{C}$ operation at the heart of our approach. The combination of G and S satisfies E and is deadlock-free. Moreover, S is guaranteed to be maximally permissive. Unfortunately, these strong guarantees do not carry over to the artifacts produced from $sup\underline{C}(G,E)$ using our approach. For instance, if our approach stops with output "fail", it is possible that a supervisor for C and E_{in} exists, because the manual inspection overlooked that, e.g., E does not correctly capture E_{in}. In addition, if the approach stops with output C_{sup}, it is still possible that C_{sup} violates E_{in}, because, e.g., the added assertions are not suitable to detect that E actually does not capture E_{in} correctly. The manual steps involved make this situation unavoidable. Moreover, since E_{in} is given only informally, it is difficult to establish theoretical guarantees with respect to E_{in}. Nonetheless, our experience suggests that the approach is still useful. During our case studies it repeatedly helped us identify inputs with unexpected, non-seeded bugs to the synthesis step. A few of these cases will be illustrated in the next section.

Also, in our experiments, we routinely found that the shared-variable implementation of the supervised code had substantially fewer states than the message-passing implementation. This suggested that the generation of the message-passing version, if necessary at all, be postponed until the very end of the prepare-synthesize-analyse cycle.

5 Working Example: Transfer-Line

We have applied our approach on several examples and used the IDES DES tool [IDE] to compute the synchronous product and $sup\underline{C}$ operations. Our working example was taken from [Won11]. A widget processing transfer-line (shown in Figure 3) consists of two production machines $M1$ and $M2$ and one test unit TU. The three machines form a production line and are connected via two widget buffers $B1$ and $B2$. $M1$ may be requested to start production of one widget at a time and deliver it to $B1$ in an unpreventable way after a arbitrary time. Similarly, $M2$ may be requested to pick-up one widget from $B1$ and then deliver it to $B2$. Finally, TU can pick up one widget from $B2$, test it and then either uncontrollably return it to $B1$ on failure or deliver it away.

Figure 4 lists the corresponding unsupervised Promela code. Code doing actual work is abstracted out with comments and the widget test in TU is replaced by a non-deterministic choice.

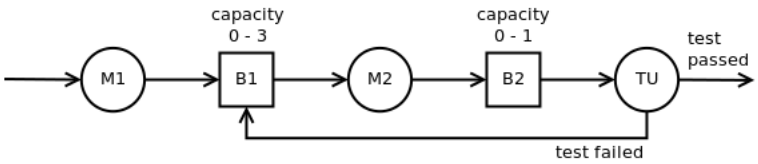


Figure 3: Transfer-Line Example

```
active proctype M1() {
  do :: true ->
    // Idle
    // Create new widget
    // Deliver widget to B1
  od; }
active proctype M2() {
  do :: true ->
    // Idle
    // Pick up widget from B1
    // Process widget
    // Deliver widget to B2
  od; }
```

```
active proctype TU() {
  do :: true ->
    // Idle
    // Pick up widget from B2
    // Test widget
    if
    :: true -> // Passed: deliver away
    :: true -> // Failed: return to B1
    fi;
  od; }
```

Figure 4: Unsupervised Promela Code C

5.1 Step 1: Preparation

Addition of Event Markup and Assertions: Since the event names chosen for the event markup in C_e will also be used for the construction of $\{E_i\}$, we start by identifying the relevant events in C and assertions suitable for checking aspects of E_{in}. The resulting code $C_{e,a}$ is shown in Figure 5. C_e is like $C_{e,a}$ except that the assertions are removed. Three controllable events (*M1MakeWidget*, *M2PickUpWidget*, and *TUPickUpWidget*) and six uncontrollable events (*M1WidgetDelivered*, *M2WidgetPickedUp*, *M2WidgetDelivered*, *TUWidgetPickedUp*, *TUWidgetPassed*, and *TUWidgetFailed*) have been identified. Event *M1MakeWidget* indicates that $M1$ is ready to produce a new widget, similarly for *M2PickUpWidget* with $M2$ from $B1$ as well as for *TUPickUpWidget* with TU from $B2$. Completed widget deliveries are signalled using *M1WidgetDelivered* and *M2WidgetDelivered* and TU signals a failed widget returned to $B1$ with *TUWidgetFailed*.

Assertions warrant the capacity constraints via auxiliary variables (`B1` and `B2`) that store the number of widgets in each buffer and model widget deliveries and pick-ups. Although not essential, the action of picking up widgets was made non-instantaneous to admit more concurrency.

Formal Specifications E_{B1} and E_{B2}: Two specifications are produced capturing how the number of elements in each of the buffer changes in response to certain events (Figure 6). Plain arrows represent uncontrollable events.

5.2 Step 2: Synthesis

Build E (Transformation T_1): The synchronous product of E_{B1} and E_{B2} was generated and contains 8 states and 58 transitions. It is not shown here due to space limitations.

Generate Plant G (Transformation T_2): Plant FSAs (Figure 7) were automatically generated from the control flow graphs of the processes in $C_{e,a}$ using standard parsing technology. Dashed

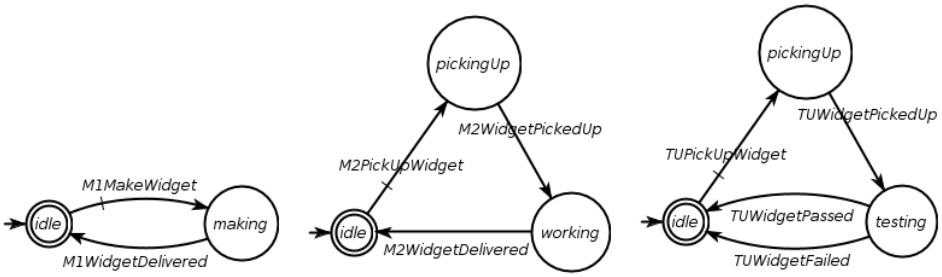
```
short B1 = 0, B2 = 0;
active proctype M1() {
  do :: true ->
    // Idle
    // relevant controllable event: M1MakeWidget
    // Create new widget
    atomic{assert(B1 < 3); B1++;}     // Deliver widget to B1
    // relevant uncontrollable event: M1WidgetDelivered
  od; }
active proctype M2() {
  do :: true ->
    // Idle
    // relevant controllable event: M2PickUpWidget
    atomic{assert(B1 > 0); B1--;      // Pick up widget from B1
    // relevant uncontrollable event: M2WidgetPickedUp
    // Process widget
    atomic{assert(B2 < 1); B2++;}     // Deliver widget to B2
    // relevant uncontrollable event: M2WidgetDelivered
  od; }
active proctype TU() {
  do :: true ->
    // Idle
    // relevant controllable event: TUPickUpWidget
    atomic{assert(B2 > 0); B2--;}     // Pick up widget from B2
    // relevant uncontrollable event: TUWidgetPickedUp
    // Test widget
    if :: true -> // Passed: deliver away
                  // relevant uncontrollable event: TUWidgetPassed
       :: true -> // Failed: return widget to B1
                  atomic{assert(B1 < 3); B1++;}
                  // relevant uncontrollable event: TUWidgetFailed
    fi; od; }
```

Figure 5: Unsupervised Code $C_{e,a}$ with Event Markup and Assertions

(a) E_{B1} (b) E_{B2}

Figure 6: Formal Specifications E_{B1} and E_{B2} (self-loops with events $M1MakeWidget$, $M2PickUpWidget$, $TUPickUpWidget$ and $TUWidgetPassed$ at each node omitted)

arrows represent controllable events. The synchronous product of $M1$, $M2$ and TU was then generated and contains 18 states and 60 transitions. It is not shown here due to space limitations.
Generate Supervisor S: The supervisor for plant G and specification E was generated with $supC$. It contains 41 states and 94 transitions. Due to space limitations it is not shown here.
Generate Supervised Code C_{sup} (Transformation T_3): We created a conversion script that implements FSAs generated by the DES tool used, and inserts concurrency control code in the original Promela code for each relevant event markup. Our script generates two distinct solu-



(a) FSA for $M1$ (b) FSA for $M2$ (c) FSA for TU

Figure 7: FSAs for $M1$, $M2$ and TU

tions: one that implements the communication between the processes and the supervisor using shared variables and another one that uses message passing.

Shared Variable Solution: For each controllable event e, a global boolean variable `_e` indicates whether e is currently enabled. Communicating the occurrence of an event to the supervisor is achieved using global variable `_Event`. When `_Event` = -1, all the events currently enabled are allowed to occur. One such event e_n (with $n \in \mathbb{N}$) is selected non-deterministically (in Spin) and its corresponding process signals its triggering by setting `_Event` to n. The supervisor indicates that it has noted and processed the occurrence of event e_n by resetting `_Event` back to -1.

During transformation T_3, for both controllable and uncontrollable events, every occurrence in the Promela source code of

```
// relevant (un)controllable event: Eventn
```

is replaced by

```
// relevant (un)controllable event: Eventn
atomic{((_Event < 0) && _Eventn) -> _Event = n;}
```

Figure 8 shows the abridged generated supervisor. The first `if` statement enables and disables all events according to the current state of the supervisor FSA. Once an event is triggered by one of the processes via global variable `_Event`, the second `if` statement realizes the corresponding transition. Note that processes can possibly block at uncontrollable events. This may be counter-intuitive, but it is required to ensure that the supervisor can process all event occurrences. However, the process will never block for long as DES guarantees that the supervisor will enable all uncontrollable events that can possibly occur after a controllable one, and therefore that it will (eventually) process any uncontrollable event to occur after a controllable one.

Message Passing Solution: Two channels are used to connect the processes with the supervisor. Channel `_EventReady` is used by processes to signal the readiness of controllable events and to indicate the occurrence of uncontrollable events. Channel `_EventGo` is used by the supervisor to trigger a controllable event (selected non-deterministically in Spin if more than one is ready).

During transformation T_3, every occurrence in the Promela source code of

```
// relevant (un)controllable event: Eventn
```

```
short _Event = 0;                          // Global mutexes
bool _Event1 = false, _Event2 = false, _Event3 = false, ...;
active proctype _Supervisor() {            // Supervisor process
   atomic { short state = 0;               // Current (and firstly initial) state
      do                                   // Main loop
         :: if                             // Enable and disable all events
               :: (0 == state) -> _Event1 = true; _Event2 = false; ...;
               :: (1 == state) -> _Event1 = false; _Event2 = true; ...;
               :: (2 == state) -> _Event1 = true; _Event2 = true; ...;
               ...                         // More cases here
            fi;
          -> _Event = -1; _Event > -1;     // Wait for an event from one of the processes
            if                             // Transition to next state
               :: ((0 == state) && (1 == _Event)) -> state = 1;
               :: ((0 == state) && (2 == _Event)) -> state = 2;
               :: ((1 == state) && (1 == _Event)) -> state = 3;
               ...                         // More cases here
            fi; od; } }
```

Figure 8: Generated Supervisor Using Shared Variables

is replaced for controllable events by

```
// relevant controllable event: Eventn
atomic{assert(nfull(_EventReady)); _EventReady ! n; _EventGo ?? n;}
```

and for uncontrollable events by

```
// relevant uncontrollable event: Eventn
atomic{assert(nfull(_EventReady)); _EventReady ! n;}
```

Figure 9 shows the abridged generated supervisor. Both channels are initially set to maximum capacity as deadlock-freedom may be lost if either channel overflows. To detect this, every send to either channel is prefixed with an "`assert(nfull())`". Both minimal capacities are determined through repeated analyses with decreasing capacities. Each event e received on `_EventReady` causes array position `eventReady[e]` to be incremented so to in effect wait on all events concurrently for a relevant event r. If event r is controllable, then r is sent back on `_EventGo` to allow the corresponding process blocked on "`_EventGo ?? r`" to proceed. The second `if` statement realizes the FSA transitions. Contrary to the shared variable solution, no process ever blocks on any uncontrollable event.

5.3 Step 3: Analysis

The analysis is used to find bugs in the formal specifications ($\{E_i\}$), the event markup (C_e), or the implementation of the transformations T_2 or T_3[3]. Simulation allowed us to locate a bug in the creation of the FSAs for the Promela processes in transformation T_2. The FSAs for $M2$ and TU did not have *M2WidgetPickedUp* and *TUWidgetPickedUp* transitions, respectively. This omission allowed $M1$ to put a fourth widget into $B1$ causing it to overflow. Verification allowed us to locate an event markup that was incorrectly placed. More precisely, event *M1WidgetDelivered* was accidentally put before `B1++` which allowed $M2$ to attempt to pick up a widget from an empty $B1$ causing the assertion `B1 > 0` in $M2$ to be violated.

[3] Since transformation T_1 just takes the synchronous product of the specifications and is assumed to be implemented using a DES tool, it is substantially simpler and is unlikely to contain bugs.

```
chan _EventReady = [255] of { byte };          // Global channels
chan _EventGo = [255] of { byte };
active proctype _Supervisor() {                // Supervisor process
   atomic { byte eventReady[10], event;        // Event buffer and variable
      do                                       // Main loop
         :: if // Find an event relevant to current state else buffer next event
              :: (0 == state) -> do
                 :: (eventReady[1] > 0) -> assert(nfull(_EventGo)); _EventGo ! 1;
                                           event = 1; break;     // Controllable
                 :: (eventReady[2] > 0) -> event = 2; break;     // Uncontrollable
                 :: else -> _EventReady ? event; eventReady[event]++;     od;
              :: (1 == state) -> do
                 :: (eventReady[3] > 0) -> event = 3; break;     // Uncontrollable
                 :: else -> _EventReady ? event; eventReady[event]++;     od;
              ...                              // More cases here
            fi;
          -> eventReady[event]--;
            if                                 // Transition to next state
               :: ((0 == state) && (1 == event)) -> state = 1;
               :: ((0 == state) && (2 == event)) -> state = 2;
               ...                             // More cases here
            fi; od; } }
```

Figure 9: Supervisor Using Message Passing

5.4 Performance Results

We also applied our method to the Dining Philosophers problem and the Cigarette Smokers Problem [Pat71]. We obtained the verification results listed in Table 1, with `ispin.tcl` and Spin Version 6.0.1[4]. We verified our three examples both with shared variables and message passing. In all cases, the following options were selected: invalid endstates and assertion violations safety checks, depth-first search, exhaustive storage mode, no compression or reduction. We also determined the minimum channel capacities. Note that for our examples, message passing requires at least 12 times more states and transitions than shared variables.

Program	Depth Reached	Stored States	Transitions	Atomic Steps	Minimum Channel Capacity Ready, Go	Number of Processes	Time to Compute $supC$ in IDES
Transfer-line							4 sec.
Shared Variables	718	1240	3207	2552	N/A	4	
Message Passing	3887	18868	47209	327715	7, 2	4	
Philosophers							1 sec.
Shared Variables	6022	10464	46033	21632	N/A	6	
Message Passing	9999	157827	580416	1326625	7, 2	6	
Smokers							1 sec.
Shared Variables	194	608	1849	904	N/A	5	
Message Passing	1996	10461	27543	82703	5, 1	5	

Table 1: Verification Results for the Three Examples

[4] A 64 bit AMD Dual Core 2.4GHz CPU with 1.5GB of DDR2 RAM was used.

6 Implementation

All our FSAs were drawn and created using a DES tool called IDES [IDE] developed by the Discrete-Event Control Systems Lab at Queen's University. Synchronous products and $sup\underline{C}$ were computed with IDES which saves its FSA files in a text XML format. Our prototype script for implementing transformation T_2 was written in Ruby and can parse most of Promela except for the `goto` statement and the newly introduced `for` statement. It takes as input a Promela text source file (C_e) and generates plant FSAs readable by IDES. Our script for doing transformation T_3 was also written in Ruby and uses the REXML XML processor. It takes as input a Promela source file (containing $C_{e,a}$), an FSA XML text file generated by IDES (containing E) and generates the supervised code (C_{sup}).

7 Conclusion

We have presented an approach which integrates DES supervisor synthesis and model checking to help facilitate the development of provably correct concurrent code. The approach recasts the process described in [DDR08] using Promela and it uses Spin for validation of the synthesis itself and the inputs to this process. We have described a prototype implementing the approach which supports shared memory and message passing concurrency and have shown how this choice can be used to optimize the verification of the generated Promela code. We have illustrated the approach with an example and provided some performance results.

Future work: There are many interesting avenues for future research. An immediate one is investigating the use of modular [WR88] and decentralized DES theory [RW92b]. Modular DES theory leverages the structure of the system and the specification to combat the explosion of the state space during the synthesis, while decentralized DES allows decentralized control by synthesizing a collection of supervisors. Ultimately, DES theory is concerned with the prevention of undesirable sequences of events. As such, it should also be applicable to other problems in software engineering. Adaptor synthesis (as in, e.g., [BBC05]) and protocol synthesis for web services (as in, e.g., [BIPT09]) are just two examples.

Finally, the development of a tool that seamlessly integrates DES theory as described here and model checking would be interesting not only for research but also for educational purposes and it would, in our opinion, represent a useful first step towards combining concepts from computer science and electrical engineering curricula as advocated in [HS07].

Bibliography

[AAE04] P. C. Attie, A. Arora, E. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Program. Lang. Syst., 26(1):125–185* 26(1):125–185, 2004.

[ADR09] A. Auer, J. Dingel, K. Rudie. Concurrency Control Generation for Dynamic Threads. In *47th Annual Allerton Conf. on Communication, Control, and Computing*. 2009.

[BBC05] A. Bracciali, A. Brogi, C. Canal. A formal approach to component adaptation. *Journal of Systems and Software* 74(1):45–54, 2005.

[BCD+07] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, D. Lime. UPPAAL-TIGA: Time for Playing Games! (Tool Paper). In *CAV07*. 2007.

[BIPT09] A. Bertolino, P. Inverardi, P. Pelliccione, M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/SIGSOFT FSE'09*. 2009.

[CL08] C. Cassandras, S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2 edition, 2008.

[Dah10] W. Dahm. US Air Force Chief Scientist Report on Technology Horizons: A Vision for Air Force Science & Technology During 2010-2030. Technical report, US Air Force, AF/ST-TR-10-01, 2010.

[DDHM02] X. Deng, M. B. Dwyer, J. Hatcliff, M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE '02*. 2002.

[DDR08] C. Dragert, J. Dingel, K. Rudie. Generation of Concurrency Control Code using Discrete-Event Systems Theory. In *FSE 16*. 2008.

[DES] DESUMA Software. Univ. Michigan and Mount Allison Univ. Avail. at www.eecs.umich.edu/umdes/toolboxes.html, last accessed March 2011.

[GPB05] D. Giannakopoulou, C. S. Pasareanu, H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engin.* 12(3):297–320, 2005.

[GPT06] A. Gromyko, M. Pistore, P. Traverso. A tool for controller synthesis via symbolic model checking. In *WODES'06*. IEEE, 2006.

[HS07] T. Henzinger, J. Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, Oct. 2007.

[IDE] IDES: The integrated discrete-event systems tool. Queens Univ. Avail. at www.ece.queensu.ca/hpages/labs/discrete/software.html, last accessed March 2011.

[IS08] W. Ifill, S. S. A step towards refining and translating B control annotations to Handel-C. In *Concurrency and Computation: Practice and Experience*. 2008.

[IST07] W. Ifill, S. Schneider, H. Treharne. Augmenting B with Control Annotations. In *LNCS*. 2007.

[Laf88] S. Lafortune. Modeling and analysis of transaction execution in database systems. *IEEE Transactions on Automatic Control* 33:439–447, 1988.

[MPS08] H. Mueller, M. Pezze, M. Shaw. Visibility of Control in Adaptive Systems. In *SEAMS 2008*. 2008.

[Pat71] S. Patil. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. Technical report, MIT, 1971.

[RW87] P. J. Ramadge, W. M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization* 25(1):206–230, 1987.

[RW89] P. J. Ramadge, W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE* 77(1):206–230, 1989.

[RW90] K. Rudie, W. M. Wonham. Supervisory Control of Communicating Processes. In *Protocol Specification, Testing, and Verification*. Elsevier, 1990.

[RW92a] K. Rudie, W. M. Wonham. Protocol verification using discrete-event systems. In *31st IEEE Conference on Decision and Control*. 1992.

[RW92b] K. Rudie, W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control* 37(11):1692–1708, 1992.

[TCT] TCT tool. Univ. of Toronto. Avail. at www.control.toronto.edu/DES, last accessed March 2011.

[TI08] M. Tivoli, P. Inverardi. Failure-free coordinators synthesis for component-based architectures. *Science of Computer Programming* 71(3):181–212, 2008.

[TMH97] J. Thistle, R. P. Malhamé, H. Hoang. Feature interaction modelling, detection and resolution: A supervisory control approach. In *Feature Interactions in Telecommunications and Distributed Systems IV*. 1997.

[WCL+10] Y. Wang, H. Cho, H. Liao, A. Nazeem, T. Kelly, S. Lafortune, S. Mahlke, S. Reveliotis. Supervisory Control of Software Execution for Failure Avoidance: Experience from the Gadara Project. In *WODES'10*. 2010.

[WKL07] Y. Wang, T. Kelly, S. Lafortune. Discrete control for for safe execution of it automation workflows. In *EuroSys'07*. 2007.

[WLK+09] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, S. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL'09*. 2009.

[Won11] W. M. Wonham. Supervisory Control of Discrete-Event Systems. 2011. Avail. at www.control.utoronto.ca/~wonham, v. 2010.07.01, last accessed April 4, 2011.

[WR88] W. M. Wonham, P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems* 1:13–30, 1988.

[ZS05] R. Ziller, K. Schneider. Combining supervisor synthesis and model checking. *ACM Transactions on Embedded Computing Systems* 4(2):221–362, 2005.