Towards Component-Based Design and Verification of a μ-Controller*

Yunja Choi and Christian Bunse

[1] School of Electrical Engineering and Computer Science
Kyungpook National University, Daegu, Korea
yuchoi76@knu.ac.kr
[2] School of IT
International University, Bruchsal, Germany
Christian.Bunse@i-u.de

Abstract. Model-driven and component-based software development seems to be a promising approach to handling the complexity and at the same time increasing the quality of software systems. Although the idea of assembling systems from pre-fabricated components is appealing, quality becomes a major issue, especially for embedded systems. Quality defects in one component might not affect the quality of the component but that of others. This paper presents an integrated, formal verification approach to ensure the correct behavior of embedded software components, as well as a case study that demonstrates its practical applicability. The approach is based on the formalism of abstract components and their refinements, with its focus being on interaction behavior among components. The approach enables the identification of unanticipated design errors that are difficult to find and costly to correct using traditional verification methods such as testing and simulation.

1 Introduction

Concerning stand-alone devices, the correctness and reliability of the relevant control systems have only limited effects. In today's new computing environments, such as ubiquitous computing and autonomous systems, however, the reliability of one, even small, embedded system may affect a large network of embedded systems. Thus, there is an urgent need for a structured development methodology with integrated verification support [13]. Model-driven Development(MDD), combined with component architecture, has been increasingly attracting researchers and industry practitioners in this regard.

Component-oriented MDD helps to cope with the increasing complexity of software system development. MDD development processes, such as Marmot [2,4] explicitly distinguish component specifications (contracts or interfaces) from component realizations (implementations). The Marmot approach

* A longer version of this paper is under review for publication in Formal Aspects of Computing.

maintains simplicity through a divide-and-conquer strategy, while keeping continuity and consistency through systematic, iterative refinements of components across different life-cycle stages (design, implementation, etc.). Furthermore, it facilitates automated formal verification as well as model-based simulation and testing, which can be naturally blended into the abstraction refinement process.

In this paper, we present a formal verification approach integrated into model driven development process. Our approach is two-fold: (1)We formalize components at each iteration of the component specification process and apply model checking [8] as a formal verification method to ensure behavioral consistency with the external environment, even before each component is completely realized. (2) We formalize gradual refinements through decomposition and verify their validity by checking behavioral consistency. While traditional verification methods, such as testing and simulation, focus on verifying *expected outputs* from *planned inputs*, model checking is based on exhaustive verification for all possible input sequences, and, thus, is better suited for ensuring high reliability and safety.

We demonstrate our approach by using a component-oriented MDD framework (based on Marmot) in the development of a mirror-control system (i.e., an embedded system controlling the movement of a car's exterior mirror). The focus is on checking the behavioral consistency and correctness of the system's μ-controller. We present a system model and formalize three important notions, namely: abstract component, realization, and refinement. These provide the basis for a systematic transformation into a formal language that can be checked. The model checker Spin [11] is used for checking interaction consistency and the essential properties of the controller, which reveals a potentially fatal problem in the original controller design – the possibility that a user request may be postponed indefinitely. We have identified the centralized control of events in the design of the μ-controller as the source of the problem by analyzing the counter-examples generated by the model checker. We show that changing each driver component into an independent and active component addresses the issue. This exemplifies the importance of formal verification in early design stages.

The remainder of this paper is organized as follows; Section 2 and Section 3 provide a brief description of the mirror control system and of using Marmot for its development. Section 4 defines the notion of abstract components and their interaction behavior. Section 5 demonstrates how components of a mirror control system can be formalized and translated into the formal specification language Promela [10], depending on the level of abstraction, and explains the effect of formal verification on the change of design. We conclude with a brief discussion in Section 6.

2 Mirror Control System

The mirror control system is an embedded system composed of electrical and mechanical components and is used to control the movement of a car's exterior mirror. The system allows a mirror to be moved horizontally and vertically into a

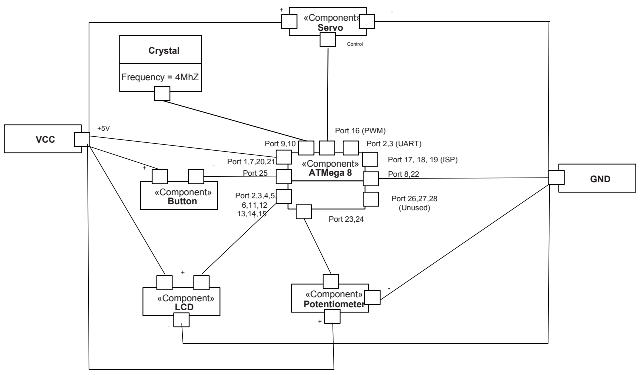


Fig. 1. UML representation of hardware

position that is convenient for the driver. Cars supporting different driver profiles can store the mirror position and recall it as soon as the profile is activated.

In the context of this paper, the mirror control system was realized in a simplified version using a μ-controller, i.e., an $ATMEL^{TM}$ $Mega$ 8, a button, and two servos (the Servo-Control System). In detail, this system requires the μ-controller to read values from the potentiometers (i.e., analog-digital conversion), converts them into the mirror turning degree, and generates the needed servo control signals, while at the same time indicating movement and degree on an LCD display. In addition, the system stores a mirror position that can be recalled by simply pressing the button. Positions are stored by pressing the button for more than five seconds. Storing and recalling is also visualized on the LCD display. Figure 1 shows a simplified UML[1] representation of the electronic circuit, representing the structural model of the mirror control system.

The requirements of the mirror control system are described by use case diagrams (Figure 2(a)) and an interaction diagram (Figure 2(b)) representing the general flow of control. The use case diagrams describe how the actor 'User' initiates the task of controlling the servo rotation. The interaction diagram provides an alternative view of the way in which user tasks are performed and shows the typical sequence of operations concerning the overall system.

3 Component-Based Development

3.1 Abstract Component and Refinements

Following the component-based development process Marmot [2,4], we view a system as a tree-shaped hierarchy of components, where the root represents the system as an abstract component, the parent/child relation represents composition, and the leaf components represent final implementation. Figure 3 illustrates

[1] The Unified Modeling Language [7].

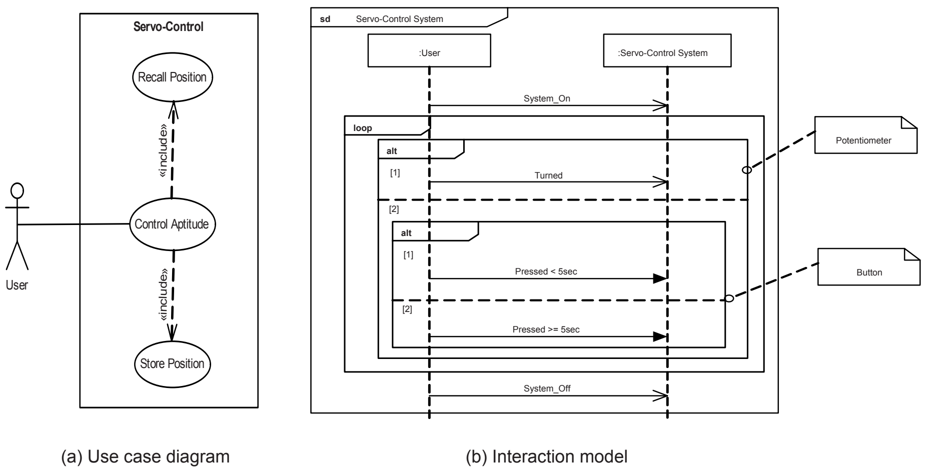


Fig. 2. Mirror control system context realization

the structure of the Marmot component divided into an externally visible contract named specification and the internal realization part; UML class diagrams and object diagrams are used to specify the external and internal structure of the component. Activity/interaction diagrams are used to specify external and internal behavior of the component. Note that we can use HDL or system C for lower-level component specifications instead of UML diagrams.

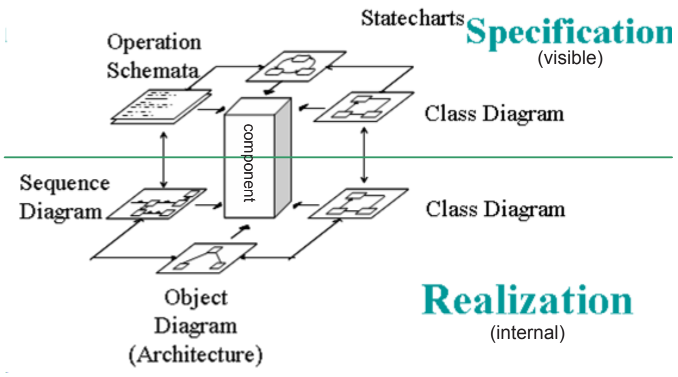
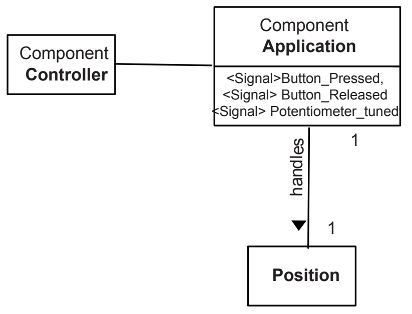


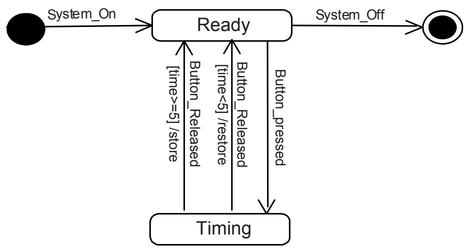
Fig. 3. Structure of a component

3.2 Component Specification

In this section, we illustrate with examples how Marmot (i.e., the MDD methodology) is applied to the development of the mirror controller. Since the hardware environment is pre-defined, we focus on the development of the software part, namely the Driver and the Application components. The Controller



(a) Application specification - class diagram



(b) Application specification - statechart

Fig. 4. Specifications for the Application component

component is a container without any software functionality, but contains the Application component. Figure 4(a) shows the specification-level class diagram of the Application component in its context. The component does not offer any operations to the outer world, but reacts to signals. This is denoted by the UML 2.0 stereotype *signal*. The state diagram of the Application component (see Figure 4(b)) shows that the Application component is in the *ready* state, after the *system_on* event. When the *button_pressed* event happens, the component transits to the *timing* state where "the time till the next *button_released* event occurs" is measured. Depending on whether the time is less than 5 seconds or not, it signals the *restore* action or the *store* action to the *Servo* component and transits back to the *ready* state.

3.3 Component Realization

While the externally visible behavior of the Application component can be uniquely specified according to the functionality required by the component, there can be various ways of realizing such functionality depending on the design decision and available hardware components that can be utilized. The MARMOT realization step specifies how the externally visible behavior of the component is actually realized through decomposition and collaboration among sub-components. Figure 5(a) shows the Driver component used to realize the functionality of the Application component, which is again decomposed into five sub-components that handle the corresponding hardware component. The interaction behavior among components is specified in an interaction diagram. For example, Figure 5(b) specifies the interaction behavior between the Application component and the button driver; the *button_pressed* (*button_released*) event from the user initiates the *Timer_starts* (*timer_stops*) action in the *button* driver, and then, depending on the duration of the button pressed event, the Application component interacts with the button driver to either set or store the mirror position in the *Servo* component.

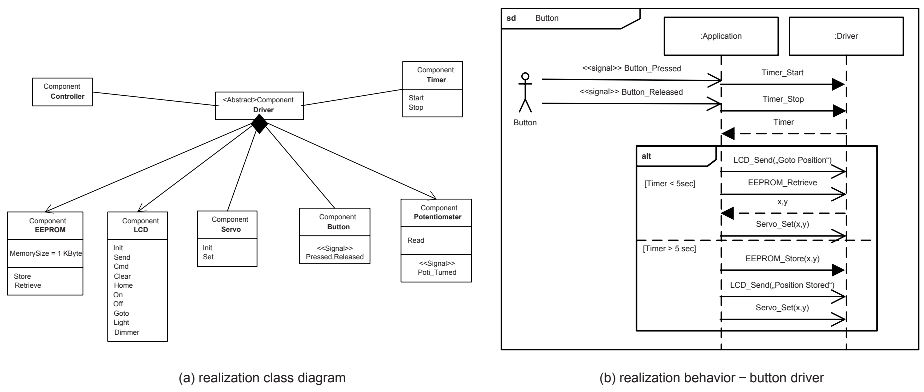


Fig. 5. Realization of the Application component

4 Formal Definition and Translation

To ensure the correctness and consistency of its complex behavior produced by the compositions of dozens, possibly hundreds, of components, we first define the meaning of component interactions and their inter-relationships using π-calculus [16], with an emphasis on their communication behavior. The formalism defined with π-calculus serves as a basis for defining translation rules from Marmot components to formal modeling languages such as Promela for verification purposes. π-calculus supports parallel composition of processes, synchronous communication between processes through channels, dynamic creation of channels, and non-determinism — characteristics suitable for formalizing abstract components and their interaction behaviors.

4.1 Formal Meaning of the Abstract Component

Figure 6 summarizes the formal descriptions for the major artifacts of a Marmot component.

We define an abstract component as a composition of two parallel processes consisting of an interface I and an externally visible body $Spec_0$ (see row 1 of Figure 6); each *Comp_spec* has pre-defined input/output channels (i, o), a set of operations *op_set*, and a set of actions *action_set* that are visible from outside the component. These are used for the component to interact with its external environment. The interface I and $Spec_0$ interchange messages and events through the internal channels u, v. Here, *new* u, v means that the channels u, v are dynamically created within the component with a limited scope. The symbol " | " represents parallel composition of two processes.

An interface $I(i, o, u, v)$ (see row 2 of Figure 6) either receives a message x from input channel i and forwards it to the internal message channel u, or receives a message y from the internal output channel v and forwards it to output channel o. Here, $i?x$, $u!x$ represents an input event on channel i, an output event on channel

	Type	Formal description
1	Abstract component	Comp_spec(i,o,op_set, action_set) = new u,v (I(i,o,u,v) | Spec_0(u,v, op_set, action_set))
2	Interface	I(i,o,u,v) = i?x.u!x.I(i,o,u,v) + v?y.o!y.I(i,o,u,v)
3	Component behavior	Spec_i(u,v,op_set,action_set) = u?x.[x=op_k].Action_spec_k(u,v,op_set,action_set) + u?x.[x!=op_k].Spec_i(u,v,op_set, action_set)
4	Abstract implementation	Action_spec_i(u,v,op_set,action_set) = (v!a_j)*.Spec_j(u,v,op_set,action_set)
5	Component realization	Comp_real(i,o,op_set,action_set) = new u,v, {(u_i,v_i)}_i (I(i,o,u,v) | !_i SubComp_i (u_i,v_i,sub_op_set_i, sub_action_set_i) | Composit_Rel(u,v, {(u_i,v_i)}_i)
6	Component Relations	Composit_Rel(u,v,{(u_i,v_i)}_i) = ∑ v_i?x.f(v_i)!x.Composit_Rel(u,v,{(u_i,v_i)}_i)

Fig. 6. Definitions of components and refinements

u, with message/signal x, respectively. Two events that are concatenated with a "." symbol occur sequentially; for example, $i?x.u!x$ means that an input event is followed by an output event. A "+" symbol means a non-deterministic choice between two different event sequences. Note that I is recursively defined so that it transits back to itself after any pair of input/output events.

The behavior of a component is defined with a series of processes from $Spec_0$, representing the process at the initial state, to $Spec_i$ representing the process at the i_{th} state, as defined in row 3 in Figure 6; $Spec_i$ receives a message x, checks whether it matches one of the operations in the *op_set*, and performs corresponding actions $Action\_spec_k$ if it matches an operation op_k, and does nothing if it does not match any of the operations in the set[2]. $Action\_spec_i$ (the row 4 in Figure 6) defines a series of actions that need to be performed by the component for a particular operation. The actions are notified to the internal output channel v, which is forwarded to the external output channel o by the interface I[3]. After all the action outputs, the process reduces to $Spec_j(u, v, op\_set, action\_set)$ where the mapping from $Spec_i$ to $Spec_j$ is predefined by the conditions on the message values. In other words, there is a mapping from $\{(i, x) \mid \textit{process state } i, \textit{ message value } x\}$ to $\{j \mid \textit{process state } j\}$. This mapping can be extracted from the statechart diagrams in the specification model.

4.2 Formal Meaning of Component Realization

Specifications of a component uniquely define the externally visible behavior of the component regardless of how it is realized internally. On the other hand, each functionality can be realized in many different ways through decomposition and refinements. The focus of this realization process is to make it as flexible

[2] The simplified notation $[x = op_i].Action\_spec_i(u, v, action\_set)$ is used instead of enumerating all possible matches.

[3] The simplified notation $(v!a_j)^*$ is used to denote a series of output actions instead of $v!a_1.v!a_2.v!a_3..v!a_n$.

as possible so that the change of a certain realization of a component does not affect the overall interaction behavior. To this end, we formally refine the abstract component *Comp_Spec* process with the *Comp_real* process, as defined in the row 5 and row 6 in Figure 6, consisting of a number of parallel *SubComp* processes that collaboratively realize the *Spec* process of *Comp_Spec*. Note that each *SubComp* is considered as an independent component on its own, and, thus, can be recursively specified as an abstract component in the same way as *Comp_Spec*.

In Figure 6 (Component Realization), $\{u_i, v_i\}_i$ abbreviates an i number of input/output channel pairs, and $!_i$ abbreviates the parallel composition of a number of *SubComp* processes whose interrelation is defined in *Composit_Rel*; for each component, the destination of its output message is uniquely defined in *Composit_Rel* in the form of a function $f : \{v, \{v_i\}_i\} \longrightarrow \{u, \{u_i\}_i\}$. This function f is used to wire sub-components and can be changed independently from the implementation of each *SubComp*, supporting flexible design for component-based development.

4.3 From Marmot to Promela

We use the model checker Spin [11] as a back-end verifier for Marmot models. There are a couple of automated verification tools directly supporting π-calculus, such as the Mobility Workbench from Uppsala University. Nevertheless, their efficiency and usability are not as good as those of general-purpose model checkers such as Spin and Smv, and, thus, they are not yet suitable for routine use during the development process.

The use of Spin requires a translation of Marmot models into Promela, the input language of Spin. The syntactic transformation from Marmot to Promela is based on the formal meaning of Marmot components defined in the previous sections. Note that our translation approach is specialized in Marmot components. For more general translations, please refer to [21].

Figure 7 shows some of the syntactic translation definitions from Marmot to Promela; the names of operations and actions in a Marmot component are translated into elements of the Promela *mtype* construct. Each communication channel in a Marmot component is declared as a message channel of *mtype* in Promela. Each Marmot component specification, component interface, and component realization corresponds to a *proctype* declaration. The Promela *run* construct is used to activate an interface process or a specification process in a component. Message sending and receiving actions can be directly translated into $u!x$ and $u?y$ where x and y are declared as *mtype*. A behavioral specification $Spec_i$ corresponds to a state of a component whose transition is defined by the transitions in $Spec_i$. A similar translation applies to $Action\_spec_i$. Non-conditional action transitions are translated into sequential actions followed by a *goto* statement. The Promela if construct is used for conditional transitions.

Note that we omit detailed translation rules from UML diagrams to Promela statements to save space. Interested readers may refer to existing approaches [9,15].

	MARMOT *construct*	PROMELA *construct*
messages	$O = \bigcup_{op\_set, action\_set} \{n \mid n \in op\_set \text{ or } n \in action\_set\}$	$mtype = \{n_1, n_2, \ldots, n_k\}$, where $n_i \in O$.
channels	new u	chan u = [1] of mtype

Processes	I(i,o,u,v)
	Comp_spec(i,o,op_set, action_set)
	Comp_real(i,o,op_set, action_set)
		proctype Interface(chan i,o,u,v)
		proctype Comp_spec(chan i,o){. . . }
		proctype Comp_real(chan i,o){ . . . }

Process Activation	Comp_Spec(i,o,O,A) = new u,v I(i,o,u,v) | Spec(u,v,O,A)	proctype Comp_Spec(chan i,o){
		chan u = [1] of mtype;
		chan v = [1] of mtype;
		run Interface(i,o,u,v);
		run Spec(u,v); }

actions	u?x
	u!y
		mtype x; u?x;
		mtype y; u!y;

states	$Spec_i(u, v, op\_set, action\_set)$	$state_i$:
transitions	$\pi.Spec_i(u, v, op\_set, action\_set)$	π; goto $state_i$;
conditionals	$u?x.[x = a]Spec_i(u, v, O, A)$	if :: u?[a] $\rightarrow$ goto $state_i$; fi;

Fig. 7. Syntactic translation from MARMOT to PROMELA

5 Applying Formal Methods

Based on the formalism introduced in the previous section, we now transform the UML representation of the Application component in Figure 4 into formal models in PROMELA. PROMELA [10] is the input language of the SPIN [11] model checker, which is widely used for software verification.

5.1 Formalizing the Specification of the Application Component

Direct Translation. First, we specify the Application component *Comp_Spec* using the formal definition for abstract components in Figure 6 as follows:

```
mtype = { system_on, system_off, button_pressed, button_released,
          poti_tuned, store, restore};

proctype Comp_spec(chan i, o){
   chan  u = [1] of {mtype};
   chan  v  = [1] of {mtype};
   run Interface(i, o, u, v);
   run  Spec(u,v);
}
```

In this specification, *mtype* declares the set of actions and operations used in the Application component. The *proctype* declaration is used to declare the component process with the name *Comp_spec* and the input, output channels i, o are declared in the signature of the component. Within the process *Comp_spec*, u, v are declared as channels with the message type *mtype*, i.e., the two internal channels are used to deliver messages/signals of actions and operations. Two parallel processes, $Interface$ and *Spec*, are activated by *Comp_spec* as its subprocesses using the keyword *run*.

The next PROMELA code shows the specification of the *Spec* process whose behavior is derived from the statechart of the Application component in Figure 4(b); the four labels, *Spec_0*(line 3), *Spec_1*(line 9), *Spec_2*(line 16), and *end_state*, represent the initial state, *ready* state, *timing* state, and the final state, respectively. The *Spec* process is initially in the *Spec_0* state and transits to the *Spec_1* state if the *system_on* signal is received. The transition from *Spec_1* occurs either to *Spec_2* or to *end_state* when the button is pressed or the *system_off* signal is received. In *Spec_2*, it non-deterministically sends out *store* or *restore* messages and transits to *Spec_1* if the *button_released* event occurs (line 19–line 22). Otherwise, it transits to *Spec_2* (line 23). Note that predicate abstraction [6] is applied to the original guarded action, "if $time < 5$ then *restore*, else if $time \geq 5$ then *store*", so that it is transformed into a non-deterministic choice of actions between *restore* and *store*; we first replace $time < 5$ with a boolean variable t transforming the guarded action into "if t then *restore*, else if $\neg t$ then *store*". Since the value of t is determined non-deterministically at this abstract level, we replace the guarded action with "non-deterministic choice between *store* and *restore*" as expressed in line 19 – line 22.

The specification for the Interface process is transformed similarly.

```
 1:   proctype Spec(chan u,v){
 2:     mtype x;
 3:      Spec_0:
 4:           u?x;
 5:           if
 6:           :: x == system_on -> goto Spec_1;
 7:           :: else -> goto Spec_0;
 8:           fi;

 9:      Spec_1:
10:           u?x ;
11:           if
12:            :: x == button_pressed -> goto Spec_2;
13:            :: x == system_off -> goto end_state;
14:            :: else -> goto Spec_1;
15:           fi;

16:       Spec_2:
17:           u?x;
18:           if
19:           :: x == button_released -> if
20:                                      :: 1 -> v!store; goto Spec_1;
21:                                      :: 1 -> v!restore; goto Spec_1;
22:                                      fi;
23:           :: else -> goto Spec_2;
24:            fi;

25:       end_state: goto Spec_0;
     }
```

Formal Consistency Checking. Once the abstract component is specified in PROMELA, we can check whether the behavior of the abstract component is consistent with its environment or not, even before we specify the actual implementation of the component. The notion of interaction consistency, which is formally defined in [4], can be informally stated as follows;

A component is consistent with its environment in its behavior if it either terminates normally or runs infinitely under the infinite sequence of stimuli generated from its environment.

Note that the negation of the interaction consistency implies a process deadlock situation, and thus, it is quite important to ensure that the initial design of a component satisfies the interaction consistency.

The specification for the environment of the Application component is derived from the use case scenarios shown in Figure 2, which can be directly transformed into PROMELA as shown below.

```
proctype env(chan in, out){
  do
  ::  out!system_on;
      do
      :: 1 ->
         if
         :: out!poti_tuned;
         :: out!button_pressed;
            out!button_released;
         fi;
      :: 1 -> break;
      od;
      out!system_off;
  :: 1 -> skip;
  od;
}
```

In this specification, the statements enclosed by *do..od* act like unconditional *while statements* in the C language; the statement repeats indefinitely as the *poti_tuned* signal or the *button_pressed* signal is generated non-deterministically.

This environment process *env* is composed with the *Comp_Spec* process, producing a system model for checking interaction consistency. The abstract Application component is verified to be consistent with its environment in this context using the SPIN verifier[4]; it took about 1 minutes and 521 M of memory for exhaustive verification, exploring 6×10^6 states and 1.7×10^7 transitions. Verification was performed on a PC with 2G Herz Pentium II processor and 2G bytes of memory.

5.2 Formalizing the Realization of the Application Component

The application component for the mirror control system is realized by a number of device drivers as illustrated in Figure 5(a). The realization behavior is specified in sequence diagrams defining which sub-components (device drivers) are used to realize a specific function provided by the Application component; an example is illustrated in Figure 5(b) for the *button* driver. Note that all the sub-components are designed as passive objects in this realization model and the Application component (the container of the driver components) acts as an active signal

[4] SPIN verifier provides an *invalid end-state* option which can be used to check the behavioral consistency between a component and its environment.

control center. Each signal passed to the Application component is identified with its source and handed to the corresponding driver depending on the source of the signal.

The next section of code shows a major part of the PROMELA specification for the realization model of the mirror control system directly translated from the realization diagrams.

```
 1: proctype ATMega(chan in1,out1, in2, out2, in3, out3, in4,
                        out4, in5, out5, tin, tout, sys_in,  sys_out){
 2:   mtype m;

 3:   hw_ready:
 4:       sys_in?m;
 5:       if
 6:       :: m == system_on ->system_state = system_on; goto driver_choice;
 7:       :: else -> goto hw_ready;
 8:       fi;

 9:   driver_choice:
10:     if
11:     :: in1?[m] -> goto button_driver;
12:     :: in2?[m] -> goto servo_driver;
13:     :: in3?[m] -> goto LCD_driver;
14:     :: in4?[m] -> goto potentiometer_driver;
15:     :: in5?[m] -> goto EEPROM_driver;
16:     :: sys_in?[m] -> goto sys_control;
17:     fi;

18:    sys_control:  ..
19:    button_driver:
20:       in1?button_pressed;          /* wait for button_pressed event */
21:       tin!set;                     /* set timer */
23:       in1?button_released;         /* wait for button_released */
24:       tin!reset; tout?m;           /* reset timer and get the timing info */
25:       if                           /* non-deterministic choice of action */
26:       :: 1 ->  out2!restore;
27:       :: 1 ->  out2!store;
28:       fi;
29:
30:     goto driver_choice;

31:   servo_driver: ..
32:   potentiometer_driver: ..
    }
```

Note that this realization model specifies the internal implementation of the system with detailed interactions among drivers and the controller; the *ATMega* process is initially in the *hw_ready* state waiting for the *system_on* signal, which initiates a transition to the state *driver_choice* (line 9–17) from which *ATMega* handles incoming messages and signals and chooses an appropriate driver. For example, if the signal is for the *button* driver, the *ATMega* process transits to the *button_driver* state, where the signal is handled as specified in the realization behavior in Figure 5(b) (line 20–30).

We have composed the *ATMega* process with the same environment model *env* illustrated in the previous section and checked for interaction consistency using SPIN to verify that the *ATMega* process is a valid realization of the mirror control system. SPIN verifies the interaction consistency on this model within 4 minutes consuming 549 M of memory, after searching 1.5×10^7 states and 3.8×10^7 transitions.

5.3 Property Verification and Design Change

One of the major purposes of formalizing a design model is to identify and verify key properties of the design and address issues related to the key properties if the verification activity reveals design errors. Once the design is formalized, we can apply automated verification for various design properties. For example, we may want to make sure that each external event *button_pressed* followed by *button_released* always has an effect on the *Servo* component, either setting or restoring the position of the mirror. This property can be formally stated in temporal logic[5] as

$$button\_pressed \,\&\&\, !\,(servo\_set \,||\, servo\_restore) \rightarrow true\, U\, (servo\_set || servo\_restore)),$$

meaning that *"for all possible execution traces, if the button is pressed and the servo is currently neither set nor restored, then the servo will be set or restored sometime in the future."*

This property is proven to be false in our realization design by the Spin verifier, which generates counter-examples showing various execution traces violating this property. For example, one counter-example shows that the system can stall without making any progress (process deadlock) if the *reset* signal to the timer gets lost in the middle of delivery. This is because the model is designed to lose additional messages if the channel is already occupied by another message. Another counter-example illustrates that there can be an infinite sequences of signals from *Potentiometer* that occupies the signal handler of the *ATMega* process all the time so that the handling of the *button_pressed* event is postponed indefinitely. This problem happens mainly because there is only one active process handling all the events and messages. If such a process is occupied by a hostile external component, the system cannot function as expected.

After careful review of the original design and the counter-examples, the realization model is redesigned to address the identified issues. We first introduce message buffers to make sure that there is no loss of messages directly affecting the system's behavior. We also make each driver component an active process that can handle events/messages on its own, instead of having one central message handler. The following illustrates how such a change in design is reflected in Promela;

```
proctype ATMega(chan in1,out1, in2, out2, in3, out3,
        in4, out4, in5, out5, tin, tout, sys_in, sys_out){
     ...
     run button_handler(in1, out1);
     run LCD_handler(in3, out3);
     run potentio_handler(in4, out4);
     ...
}

proctype button_handler(chan in, out, tin, tout){
     mtype m;
```

[5] Temporal logic can be considered as propositional logic with the notion of relative time. Spin is equipped with an automated verification facility for properties written in temporal logic LTL [18].

```
     button_driver:
        in?button_pressed;
        tin!set;
        in?button_released;
        tin!reset; tout?m;
        if
        :: 1 -> out!restore;
        :: 1 -> out!store;
        fi;
        goto button_driver;
}
```

Note that the *ATMega* process is simplified containing only the statements initiating different drivers whose behavior is specified as an active and independent process. The drivers are connected to the *ATMega* process by wiring them with message channels. For example, when the *button_pressed* event arrives in channel *in*1, the *button_handler* process directly recognizes this event without going through the *ATMega* process and handles it as specified in the button driver. The behavioral specification for the *button_handler* process is the same as the one specified under the label *button_driver*(line 19–30) in the previous design. With this modified design, the same property is verified to be true[6].

6 Discussion

The use of formal methods in embedded systems has been an active research issue for almost a decade [14,23], but, unfortunately, we have not seen active practice of formal methods in industry, mainly due to lack of experience, supporting tools, and methodologies [13]. We believe this situation can be altered by integrating formal methods into existing development methodologies so that the application of a formal method can be seen as a routine task within the process. As demonstrated in this paper, the use of a structural methodology makes the application of formal methods simpler and easier by providing gradual yet seamless transitions from the early design to actual implementation. Our approach is partially automated by reusing the Marmot-Promela prototype translation tool introduced in [4].

There have been other approaches that apply formal methods in embedded systems; for example, [20] uses a variation of Petri Net as the underlying formalism of a system model and translates it into Promela to use the Spin verifier. Nevertheless, this approach and other related previous approaches [12,17,23] lack an association with development methodology. Several approaches have tried to address behavioral properties in system development [1,3,5,19,22,24], where some of them use model checking for checking properties of UML diagrams [1,24]; among them, [22] is the closest to our approach in the sense that their approach is closely coupled with a component-based system development process. Nevertheless, [22] takes a bottom-up approach by identifying properties for each component under environmental assumptions. Compositional verification is performed by cleverly assembling those properties of each sub-component that have

[6] It is verified to be true under fairness constraints that all the processes are executed infinitely often.

been already verified. On the other hand, our approach extracts environmental constraints from the internal behavior of the refined component, which is specified during the Marmot refinement process, eliminating the need for manually identifying environmental assumptions [4].

Our approach emphasizes that the use of a structured development methodology such as MDD is necessary for achieving a high-quality system, but is not sufficient for it. While structured methods can cope with structural complexity using the well-known "divide-and-conquer" principle, the interaction complexity among decomposed parts of the system tends to get higher, which becomes a major problem. We tackle this particular problem with formal methods integrated into the development methodology.

We note that our approach presented in this paper is work in progress that requires further investigation on its practical aspects, especially with respect to usability and efficiency. We need more industrial case studies to claim that our approach is actually practical. There are other issues to be considered in the design of embedded systems, such as energy consumption, timing issues, and utilization of limited memory [13]. We plan to investigate such issues within the same verification frame in future work.

References

1. Adamek, J., Plasil, F.: Component composition errors and update atomicity: Static analysis. Journal of Software Maintenance and Evolution: Research and Practice (September 2005)
2. Atkinson, C., Bayer, J., Bunse, C., et al.: Component-based Product Line Engineering with UML. Addison-Wesley Publishing Company, Reading (2002)
3. Barros, T., Henrio, L., Madelaine, E.: Behavioural models for hierarchical components. In: International SPIN Workshop on Model Checking Software (August 2005)
4. Choi, Y.: Checking interaction consistency in MARMOT component refinements. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 832–843. Springer, Heidelberg (2007)
5. Engels, G., Kuester, J.M., Groenwegen, L.: Consistent interaction of software components. Journal of Integrated Design and Process Science 6(4), 2–22 (2003)
6. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
7. Object Management Group. UML2.0 superstructure specifications
8. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking: History, Achievements, Perspectives. Springer, Heidelberg (2008)
9. Guelfi, N., Mammar, A.: A formal semantics of timed activity diagrams and its PROMELA translation. In: 12th Asia-Pacific Software Engineering Conference (2005)
10. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall Software Series (1991)
11. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Publishing Company, Reading (2003)
12. Hsiung, P.-A.: Formal synthesis and code generation of embedded real-time software. In: 9th International Symposium on Hardware/Software Codesign (April 2001)

13. Johnson, S.D.: Formal methods in embedded design. IEEE Computer (November 2003)
14. Kern, C., Greenstreet, M.: Formal verification in hardware design: A survey. ACM Transactions on Design Automation of E. Systems (April 1999)
15. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.: Implementing statecharts in PROMELA/SPIN. In: Second IEEE Workshop on Industrial Strength Formal Specification Techniques (October 1998)
16. Milner, R.: Communicating and Mobile Systems: the π-calculus. Cambridge University Press, Cambridge (1999)
17. Naeser, G., Lundqvist, K.: Component-based approach to run-time kernel specification and verification. In: 17th Euromicro Conference on Real-Time Systems (2005)
18. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. Foundations of Computer Science, pp. 46–57 (1977)
19. Reussner, R.H., Poernomo, I., Schmidt, H.W.: Reasoning about software architectures with contractually specified components. In: Component-Based Software Quality: Methods and Techniques, State-of-the-Art Survey (2003)
20. Ribeiro, O.R., Fernandes, J.M., Pinto, L.F.: Model checking embedded systems with PROMELA. In: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (2005)
21. Song, H., Compton, K.J.: Verifying pi-calculus processes by promela translation. Technical report, Department of Electrical Engineering and Computer Science, University of Michigan (2003)
22. Xie, F., Browne, J.C.: Verified systems by composition from verified components. In: Proceedings of Joint Conference ESEC/FSE (2003)
23. Yang, W., Moo-Kyeong, Kyung, C.-M.: Current status and challenges of soc verification for embedded systems market. In: IEEE International Conference on System-On-Chip (2003)
24. Zimmerova, B., Brim, L., Cerna, I., Varekova, P.: Component-interaction automata as a verification-oriented component-based system specification. In: Workshop on Specification and Verification of Component-Based Systems (2005)