Software & Systems Modeling manuscript No.
(will be inserted by the editor)

Yunja Choi · Christian Bunse

Design Verification in Model-based μ-Controller Development using an Abstract Component

the date of receipt and acceptance should be inserted later

Abstract Component-based software development is a promising approach to controlling the complexity and quality of software systems. Nevertheless, recent advances in quality control techniques do not seem to keep up with the growing complexity of embedded software; embedded systems often consist of dozens to hundreds of software/hardware components which exhibit complex interaction behavior. Unanticipated quality defects in a component can be a major source of system failure. To address this issue, this paper suggests a design verification approach integrated into the model-driven, component-based development methodology MARMOT.

The notion of abstract component – the basic building block of MARMOT – helps lift the level of abstraction, facilitates high level reuse, and reduces verification complexity by localizing verification problems between abstract component *before* refinement and *after* refinement. This enables the identification of unanticipated design errors in early stages of development.

This work introduces the MARMOT methodology, presents a design verification approach in MARMOT, and demonstrates its application on the development of a μ-controller-based abstraction of a car mirror control system. An application on TinyOS shows that the approach helps reuse models as well as their verification results in the development process.

This paper is an extended version of [11,13].

This work has been supported by the Korea Research Foundation Grant funded by the Korean Government(KRF-2008-331-D00525) and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), grant number R11-2008-007-03002-0.

Yunja Choi
School of Electrical Engineering and Computer Science, Kyungpook National University, Daegu, Korea
E-mail: yuchoi76@knu.ac.kr

Christian Bunse
School of IT, International University in Germany, Bruchsal, Germany
E-mail: Christian.Bunse@i-u.de

Keywords Abstract component, Model-Driven Development, Design verification, Embedded systems

1 Introduction

Embedded software controlling electronic systems is everywhere in modern society from large-scale aircraft control systems to small-scale sensor nodes. Many of such systems are safety-critical; for example, any potential cause of failures in air-traffic control systems, medical device control systems, and braking systems in modern cars must not be tolerated. Thus, software development approaches for such systems have to specifically address quality issues and provide means to either avoid or identify them early during development [32].

One approach to address these issues is component-based development, following the ideas of Clemens Szyperski [46]: Component-based software development (CBSD) or component-based software engineering (CBSE) is concerned with the assembly of pre-existing software components into larger pieces of software. In this sense, a software component is a unit of composition with contractually specified interfaces and explicit context dependencies. Componentizing software had been suggested as a way of tackling the software crisis. In contrast to the ideas of modular development, CBSE combines elements of software architecture, modular software design, software verification, configuration and deployment.

This study suggests a design verification approach integrated into the model-driven, component-based development methodology named MARMOT [4,11]. MARMOT[1] aims at providing the ingredients to master the multi-disciplinary effort of developing embedded systems. In general, MARMOT is not meant to be a method for Hardware-Software Co-design. The focus on MARMOT is clearly on the software side of development but provides

[1] MARMOT stands for Method for component-bAsed RealtiMe Object-oriented development and Testing.

means for developing platform specific software systems. It thus, provides templates, models, and guidelines for the software products describing a system, and how these artifacts are built up throughout development. In detail, MARMOT iterates from the design of abstract components and the specification of their externally visible behavior. These abstract components are then successively refined and decomposed into sub-components in order to realize them according to the development platform and available resources. Following this view hardware components are "specified" regarding implementation-relevant aspects but MARMOT does not provide a means for their development.

The iterative nature of MARMOT helps manage the increasing interaction complexity among components, but also provides a means for pre-checking a system's behavior, even before each component has been completely realized. By focusing on the essential properties of the system, the initial abstract specification can be relatively simple, which makes it feasible to apply formal methods [15,30], including the use of automated formal verification techniques such as model checking [14,27]. The gradual refinements and decomposition is supported by the notion of abstract component[2].

This work introduces a consistency model for checking interaction consistency among *abstract components*. An abstract component is a basic building block of the MARMOT that enables high-level reuse and systematic transformation of the MARMOT model into formal specifications, suitable for performing automated formal verification. The semantics of inter-communicating abstract components is defined using the π-calculus [39] notation and their internal behavior is defined in terms of labeled transition systems. This enables the definition of a translation rule between the abstract component and formal specification language for model checking [14]. The consistency model is systematically extracted from MARMOT abstract components and verified/refuted using the model checker SPIN [27] during the iterative design process.

The proposed approach is demonstrated with a small case study by applying MARMOT during the development of an abstract version of a mirror control system, showing that potentially fatal communication problems can be identified during the iterative process of component refinement and verification. A more extensive application of the approach is being conducted on the design verification of TinyOS – a representative operating system for wireless sensor networks.

Recently, there have been several approaches for component-based development and verification [25,47, 28]. Nevertheless, this work is the first to incorporate verification activities into the entire design process, from conceptual design to physical components. The following summarizes the major differences between this work and existing approaches:

1. it is process oriented : formal verification techniques are used to support the iterative specification-refinements process. The main goal is to assure the continuity of the design process by checking the behavioral consistency of abstract components before and after their refinements.
2. it lifts the level of abstraction : The same notion of abstract component is used throughout the design process from specifying system context to generating program code. By conceptualizing the design process with a uniform notion, it facilitates systematic development and verification as well as a high-level of reuse.
3. it reduces verification complexity using a structured approach : consistency models are automatically generated from the structured information on abstract components and their refinement relationship. This information helps reduce the verification complexity by systematically localizing it to the problem among directly related abstract components in the refinement/decomposition hierarchy.

The remainder of this paper is organized as depicted in Figure 1: Section 2 discusses related work. Section 3 provides a brief overview of the MARMOT methodology. Section 4 introduces a case example, the mirror control system. Section 5 provides a brief overview of the proposed design verification approach followed by underlying formalism (Section 6), its application to mirror control system (Section 7), and experiments on a TinyOS (Section 8). This paper concludes with a discussion in Section 9.

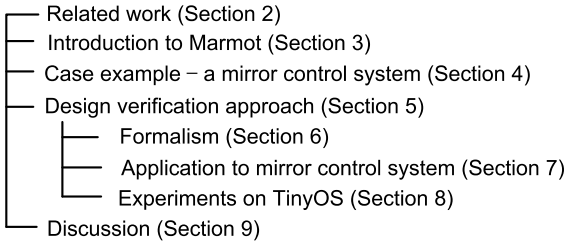


Fig. 1 Organization of this paper

2 Related Work

The component paradigm promises to address many of the productivity and quality problems currently faced by the software industry, but its correct application requires systematic and methodological verification support. This

[2] The term abstract components, coined by Braun and Wallnau [6] refers to view components as application-specific core assets, and thus emphasizes component-based design approaches rather than standard component infrastructures or component marketplaces.

chapter surveys existing approaches to component based development from two different perspectives: from the design and modeling as well as from the verification perspective.

2.1 Model-Driven and Component-based Development

A wide range of theoretical and practical methods have been developed in the context of the component paradigm. Following the taxonomy published in [8], the following major efforts in this regard can be identified.

Catalysis [16] is one of the first methods developed to leverage the UML(Unified Modeling Language) [24] in connection with component based development, which embraces many of the other reuse technologies. The method either introduced or popularized many of the ideas that today are considered natural ingredients of component-based development. Catalysis uses an iterative and incremental process based on abstraction and refinement mechanisms. These mechanisms are applied throughout system development from early analysis to implementation and set up the basis for recursive relationships between models, which then support forward- and re-engineering of systems. Catalysis makes use of the UML with strong semantic consistency and completeness criteria based on a small set of core constructs.

Select Perspective [20] emphasizes the importance of business process modeling and follows a clean process that transforms system-independent business processes into implementation-oriented models. This includes the explicit identification of components, as well as the potential integration of legacy systems. Select Perspective defines most of the essential ingredients needed for component-based development in the early stages of the software life cycle. Unlike other methods, it also explains the role that component technology can play in integrating legacy systems into new applications. However, its main weakness is that it is not always clear which aspects of the underlying business objects are being described by which models. In other words, the distribution of the information describing a business object is somewhat arbitrary.

UMLComponents [10] focuses on the specification of components using the UML. The method identifies two main phases: the requirements work-flow which captures the basic needs that the system must fulfill in terms of use cases and high level business classes, and the specification work-flow which documents the business types, interfaces, and components that have been chosen to satisfy these requirements. In essence, UMLComponents packages are a core subset of Catalysis concepts. Unfortunately, it loses key ideas such as the nesting of components to arbitrary depths, the recursive application of development concepts, and the use of frameworks to package larger-grained reusable structures than interfaces and components.

In summary, the taxonomy showed that methodological support for component-based software development has made significant advances, compared to the early methods, such as OMT(Object Modeling Technique). However, the methods available today are not silver-bullets and are sensitive to the requirements of different domains and system types. More specifically, the area of safety-critical systems requires formal development and support for addressing non-functional properties. This warrants "new" methods, such as Marmot.

2.2 Formal design verification

Existing formal design verification approaches can be categorized into three: (1) research on formal semantics for component and composition in general [2,3,48], (2) transformational approaches for the verification of Uml diagrams or architecture [5,9,35,40,44], and (3) integrated approaches in software development [1,18,28, 49].

Research on formal semantics aims at providing formal semantics for component models. For example, [48] formalizes behavioral consistency among components with respect to Petri-net semantics. [2] provides a formal notation and theory for the architectural connection of components including the notion of ports, connect, and refinement. [3] formalizes the component composition using a channel-based coordination model. These approaches mainly focus on the formal definition of a component and of composition without properly addressing practical aspects (e.g., integration into existing development methodologies and/or support by automated checking techniques).

Transformational approach transforms semi-formal models into verifiable formal models. [44] uses a variation of Petri Nets as the underlying formalism of a system model and translates it into Promela to use the Spin verifier. [5,35] define operational semantics for Uml statechart and its translation into Promela, the input language of the Spin model checker, based on the operational semantics. [40] defines a process algebra for regular sequence diagrams. The proposed approach includes these transformational approaches. Some of them [35,51] are adopted in our Uml-Promela translation.

Integrated approach integrates transformational approaches into development process. For example, the OMEGA project [28] is a notable approach for formal design verification of embedded software. The approach is based on a Uml profile that supports automated formal verification. The work demonstrates that Uml, with well-defined semantics, can be a practical choice as a modeling language for both development and verification. Nevertheless, it does not particulary address issues related to engineering methodology, such as component identification, decomposition (as well as the verification of its refinements), and reuse in the development process.

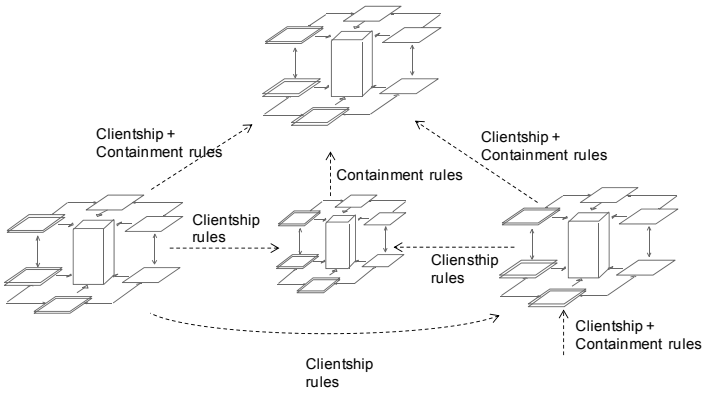


Fig. 2 Component Hierarchy

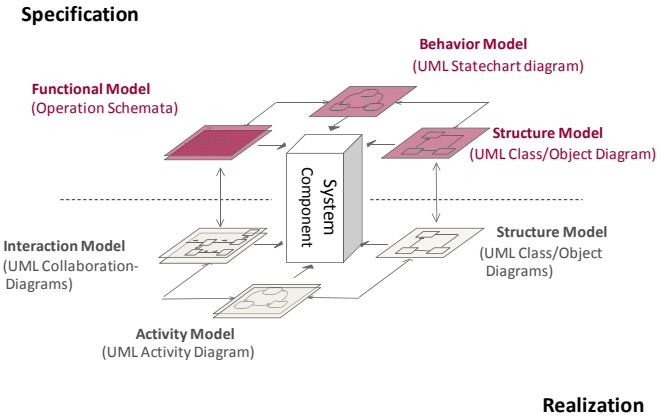


Fig. 3 Marmot Component

[49] is the closest to the proposed approach in the sense that it is closely coupled with a component-based development process. It 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 has already been verified. In contrast, the proposed approach extracts environmental constraints from the internal behavior of an abstract component, which is specified during the Marmot refinement process, eliminating the need for manually identifying environmental assumptions.

Besides, a number of direct applications of formal verification on embedded software [19,21,29,44] exist. For example, [42] models a hardware-implemented run-time kernel using Uppaal and verifies the absence of deadlocks, the correctness of kernel calls and the FIFO order using the Uppaal model checker. Nevertheless, these approaches are more toward showing the capability of automated formal verification than addressing methodological issues.

3 The Marmot Methodology

Reuse is a key success factor in today's software industry and can be regarded as a major driving force in hardware and software development. Reuse is pushed forward mainly by the growing complexity of software systems. This section introduces a methodology for the component-based development of embedded systems, referred to as Marmot, that is specifically geared towards facilitating reuse in embedded systems development. Marmot is an extension to the KobrA method [4], a component-based development framework for information systems, and adds concepts addressing the specific requirements of embedded system development.

3.1 Principles

Marmot [4,11] advocates composition as the single most important engineering activity by viewing a system as a tree-shaped hierarchy of components, in which the parent/child relationship represents composition (Figure 2). A Marmot project is based upon the following fundamental activities: (1) Iteratively decompose the system into finer-grained parts that are individually controllable (this is termed "decomposition"), and (2) reduce the level of abstraction to create representations of the system that come closer and closer to executable formats, which is termed "refinement".

Another important principle in Marmot is the separation of concerns: Separation and clear distinction of what a software unit does (e.g., "specification", "interface", and "signature") from how it does it(e.g., "realization", "design", "architecture", "body", and "implementation"). This facilitates a "divide and conquer" way to cope with system complexity and increases flexibility and reusability by allowing new versions of a unit to be easily interchanged with old versions, provided they share the same "specification".

A component modeled according to these principles (see Figure 3) is essentially described on two levels of details - one representing a component's interface (what it does) and the other representing its body (i.e., how it fulfills the specified interface). Following these principles, each component of a system can be described by a suite of UML diagrams as if it were an independent system in its own right. The notion of an *abstract component* will be described in detail later.

In principle, many methods change the way they represent certain abstractions or concepts based on the level of granularity or phase of development, at which they are being addressed, not because of any inherit changes in the concept itself. In other words, they represent and manipulate a given abstraction in distinct ways in different parts of the method. Marmot therefore follows the principle of uniformity. It requires that, wherever possible, a

given fundamental concept is represented and manipulated in the same way in all parts of the method, and in all phases of a project that uses the method.

MARMOT's approach to modeling is based on the basic idea that the development time artefacts (e.g. UML diagrams) should be constructed and organized in a way that reflects the run-time component-oriented structure of the system. This idea is captured by the so-called "principle of locality" which requires that every development artefact, including UML models and diagrams, be focused towards the description of a single run-time artefact. In other words, there are no global models or diagrams. Instead, every model or diagram is focused on the description of a single artefact: that is, it is "local" to that artefact.

A potential disadvantage of role modeling to the extent used in MARMOT is the unnecessary duplication of information. Since separate development artefacts, such as UML diagrams, necessarily overlap to some degree, and each describes a specific piece of information from its own viewpoint, there is a danger of unnecessary redundancy. This is particularly so for class diagrams which are notoriously difficult to finish, and can in principle be cluttered with an unlimited number of artificial associations. To avoid this problem, MARMOT requires every diagram to contain the minimum information needed to convey the required ideas.

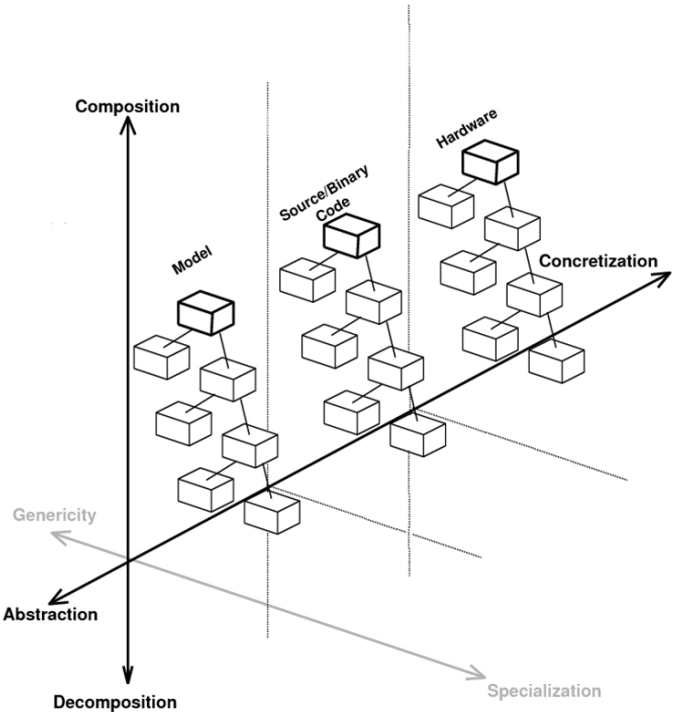


Fig. 4 Development Dimensions

3.2 Process Model

The MARMOT process model identifies four essential goals that should be achieved by any software development process. This process model prescribes generic processes to achieve each of these goals as well as a sequence in which these goals should be achieved. The core principle of MARMOT is the separation of concerns, so MARMOT associates its main development effort with two basic dimensions that are mapped to four basic activities [4]. These are depicted in Figure 4:

1. **Composition/Decomposition dimension.** Decomposition follows the "divide-and-conquer" paradigm, and is performed to subdivide the entire embedded system into smaller parts that are easier to understand and control. Composition represents the opposite activity, which is performed when the individual components have been implemented, or some others reused, and the system is put together.
   - **Decomposition.** Development projects that aim at developing a new product typically start above the top left-hand side box in Figure 4. The box represents the entire system to be built. Before the specification of the box, the concepts of the domain or the physical world in which the system is supposed to operate have to be determined. This comprises descriptions of all entities relevant to the domain, including standard hardware components that will eventually appear on the right-hand side towards concretization. In embedded systems, these implementation-specific entities often determine the way in which a system is divided into smaller parts. During decomposition, newly identified logical parts of the system are mapped to existing components. Whether these are hard- or software does not play a role during this early phase because of the way all components are treated in terms of collections of descriptive artifacts, that are, models.
   - **Composition.** After having implemented some components and having reused some others, the system can be assembled according to the abstract model. Therefore, the subordinate boxes with their respective super-ordinate boxes have to be coordinated in a way that follows the component standard described earlier exactly.
2. **Abstraction/Concretization dimension.** This dimension is concerned with the implementation of a system and the gradual refinement of artifacts towards more concrete/executable representations. The activity is called embodiment, and it turns the abstract system represented by models into more concrete representations that can be executed by a computer. The reverse direction is called verification. This activity checks whether the concrete representations are in line with the abstract ones.

– **Embodiment.** During decomposition, the shapes of each identified individual component are defined in an abstract and logical way. The system, or parts thereof, can then be moved towards more concrete representations. This means they become platform-specific.
– **Verification.** In general, software verification provides objective evidence that the result of a particular development phase meets all specified requirements for that phase. Therefore, as a final activity, verification (i.e., in reverse to embodiment) is carried out in order to check whether the concrete composition of the embedded system corresponds to its abstract description.

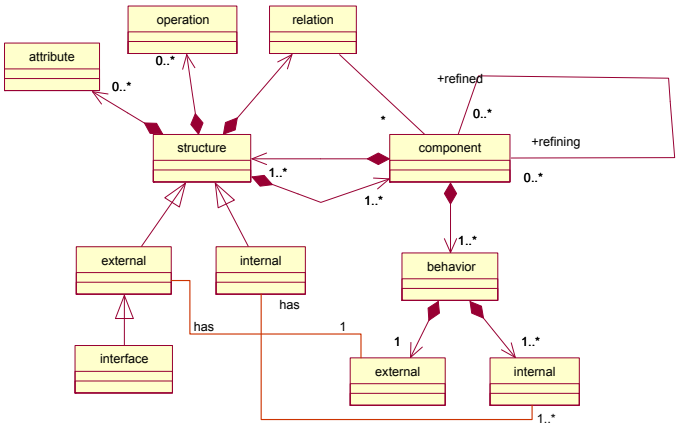


Fig. 5 MARMOT component meta-model

3.3 Product Model

The product model provides semantics and syntax for modeling software products. When using MARMOT, components and systems (i.e., in the form of a component containment tree) are built on the same fundamental principles as in object technology. Therefore, MARMOT components follow the principles of encapsulation, modularity, and unique identity that most component definitions put forward, which lead to a number of obligatory properties:

– Composability is the primary property of a MARMOT component, and can be applied recursively: Components are composed of components, which are again composed of components, etc.
– Reusability is the second key property, and can be separated into: *Development for reuse*, which deals with how components have to be specified and treated, so that they can be reused, and *development with reuse*, dealing with the integration and adaptation of existing components into a new application.
– Unique identities require that a component may be uniquely identifiable within its development environment as well as within its runtime environment. MARMOT provides the principles for that.
– Modularity/encapsulation refer to a component's scoping property as an assembly of services, which is also true for a hardware component, and as an assembly of common data, which is true for the hardware and the software parts of an embedded component. Here, the software only represents an abstraction of the hardware, which essentially provides the memory for the data.
– An additional important property is communication through interface contracts, which becomes feasible in the hardware or embedded world through typical software abstractions. Here, the additional hardware wrapper of MARMOT realizes that the typical hardware communication protocol is translated into a typical component communication contract.

Composition along the Composition/Decomposition dimension turns a MARMOT project into a tree-shaped structure with consecutively nested abstract component representations. Such a tree is called a containment tree. Every box in the tree, each representing a component or a system in its own right, is made up of a component specification and a component realization. The specification is a suite of descriptive artifacts that collectively define everything externally knowable about a component. These descriptions fully specify a component in a way that it can be assembled in a system and used by the system. The realization is a suite of descriptive artifacts that collectively define how a component is internally realized. According to the composition principles, components can be made up of other components. Any component in a MARMOT containment tree can therefore be a containment tree in its own right, and, as a consequence, another MARMOT project.

3.4 Abstract component

The MARMOT model can be uniformly represented with *abstract component*. As depicted in Figure 5, an abstract component consists of one or more *structures* and one or more *behaviors*. Each structure and behavior has an *internal* part hidden from outside the component and an *external* part visible from outside which also defines an interface of the component. The structure may contain specifications of attributes and operations of the component as well as the specifications regarding relationships with other components.

Each external structure has an external behavior visible from outside; an example can be the change of its attribute values (states) according to the service requested. Each internal structure can have one or more internal behaviors specifying how externally visible operations (services) of the component (specified in the external structure) are internally realized. If the component is decomposed into sub-components, the component acts as

a container of the sub-components and its internal behavior can be considered to specify the glue-code for the sub-components. Otherwise, an internal behavior specifies an algorithm for implementing an externally visible service. This realization can be self-contained in the component or implemented by using services from other components. In the latter case, only the services provided by *visible* components can be used; a component A is visible to a component B only if A is a refining component of B and if this relationship is explicitly specified in the internal structure of the refined component.

This abstract component can be realized using UML [24] diagrams in the software modeling stage; for example, UML class diagrams and object diagrams are used to specify the external and the internal structure of the component. Statecharts and activity/interaction diagrams are used to specify the external behavior and the internal behavior of the component, respectively[3]. For lower level component specifications, such as hardware components, one may use other modeling languages such as HDL or SystemC.

3.5 Summary

In summary, the core of MARMOT is represented by two major activities: Component Identification and Component Realization. Once a component is identified with its set of services and externally visible behavior, its realization (implementation) can be achieved either by reusing services provided by existing components or by implementing its service from scratch, which again can result in building sub-components.

In more detail, a MARMOT project is based upon the following fundamental activities:

1. Decompose the system into finer-grained parts that are individually controllable.
2. Reduce the level of abstraction to create representations of the system that come closer and closer to executable formats.

In other words, every system is organized as a tree-shaped hierarchy of logical building blocks that have class-like and package-like properties. The class-like properties allow a component to have attributes, operations and behavioral features, whereas the package-like properties allow a component to represent a name space and act as container for a wide range of documents, concepts and other components.

Figure 6 shows how the primary component engineering activities, when visualized in connection with the hierarchic product they generate can be regarded as leading to a spiral-based process. The final goal of the component reuse activities is to fully integrate a component that has been developed earlier outside the context of the tree (i.e., an external component). To achieve this, the specification desired of the reusing component and the provided specification, offered by the pre-existing component, have to be brought into agreement. When such a situation exists, the reused component realizes, and usually also implements the specification that is required by the reusing component, and the reused component is then fully integrated.

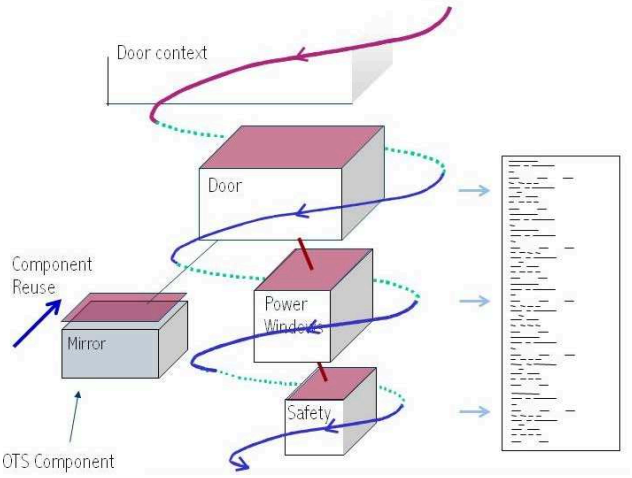


Fig. 6 MARMOT Process

4 A Case Example: Mirror Control System

The mirror control system is an embedded system composed of electrical and mechanical components and is used for controlling the movement of a car's exterior mirror (see the left side of Figure 7). The system allows the mirror to be moved horizontally and vertically into a position that allows the driver to watch traffic behind him/her. In addition, the system supports storing/recalling different mirror positions (i.e., needed for cars that make use of driver profiles).

4.1 SYSTEM overview

Since the focus of this study is on software development, the mirror control system was realized in a simplified (hardware) version using a μ-controller (i.e., an ATMEL^{TM} Mega 8), a button, and two servos, also known as the Servo-Control System (see the right side of Figure 7).

A servo drive receives a command signal from a control system, amplifies the signal, and transmits electric current to a servo motor in order to produce motion

[3] To facilitate specifying correct and unambiguous system behavior, this study uses a template for specifying operations and attributes of a component as well as an action language similar to OMAL [43] for unambiguously specifying behaviors in the UML diagrams.

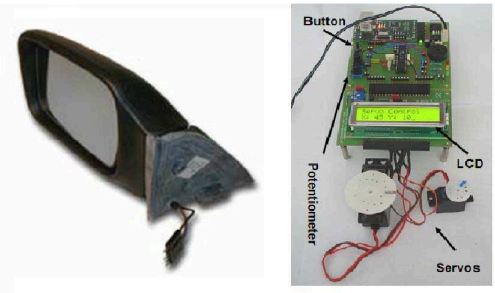


Fig. 7 Exterior mirror and its prototype realization

proportional to the command signal. Typically, the command signal represents a desired velocity. A velocity sensor attached to the servo motor transmits the actual motor velocity to the servo drive. The servo drive continually compares the actual motor velocity with the commanded motor velocity to generate an output to the motor that is to correct any error in the velocity.

The micro-controller has limited performance and resources (i.e., 4 MHz, 8KB programmable Flash memory, 512 Bytes EEPROM, and 1KB Internal SRAM) as well as minimal I/O capabilities (i.e., three counters, three PWM Channels (pulse width modulation), and a multiplexed 8-channel ADC(analog-digital conversion)).

The actual system requires the μ-controller to read values from the potentiometers (requires an analog-digital conversion), converts these readings into a turning angle, and generates the needed servo control signals (which requires PWM signal generation using timers and interrupts), while at the same time indicating movement and turning angle on the LCD. In addition, the system can save a position (turning the angles of both servos) by pressing the button for more than five seconds. The position can be later recalled by simply pressing the button for less than five seconds. Storing and recalling are visualized on the LCD. The right side of Figure 7 shows the prototype of the system realized using the MyAVR^{TM}-Board.

4.2 Software realization

The requirements of the mirror control system are described by UML usecase diagrams (Figure 8(a)) and an UML interaction diagram (Figure 8(b)) representing the general flow of control. The usecase diagram describes how the actor 'User' initiates the task of controlling the servo rotation. In addition to the graphical depiction, every usecase is also textually specified in order to capture details, not necessarily contained in the UML diagram.

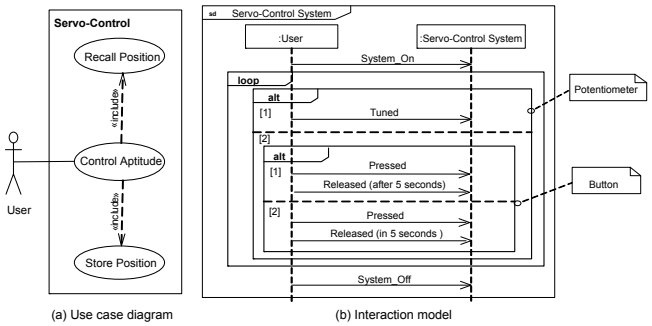


Fig. 8 Mirror-control system context realization

The interaction diagram provides an alternative view of the way in which user tasks are performed and shows the typical sequence of operations of the overall system. In addition, the diagram specifies the signals that, created by user events, are sent from the micro-controller to the software system.

Figure 9 represents the structural model of the mirror control system. Electronic components are mapped to UML classes marked by the stereotype "Component", to indicate their nature, and (optionally) "Hardware", to distinguish them from later driver classes using the same name. In order to ensure consistency, software components addressing hardware functions (i.e., driver components) are named according to the controlled hardware component. The diagrams were manually translated from the circuit diagram using Marmot mapping rules and guidelines.

In general, all these specifications form the 'context realization' of the mirror-control system. In a sense, the "context" can be viewed as a pseudo component at the root of the development tree. The system is then treated as an abstract component, since the context provides the encapsulating realization against which it can be specified. Using this information allows defining a preliminary containment hierarchy of the system. This hierarchy specifies how coarse-grained components are "made

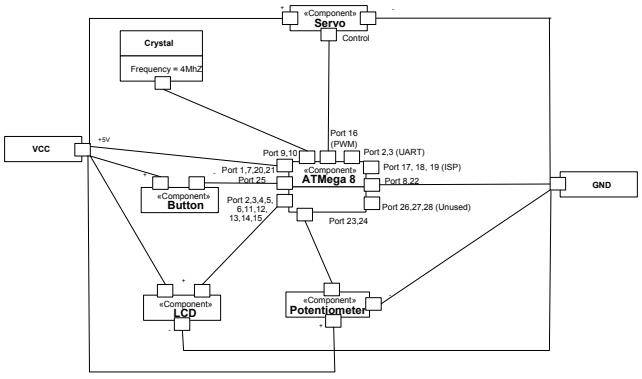
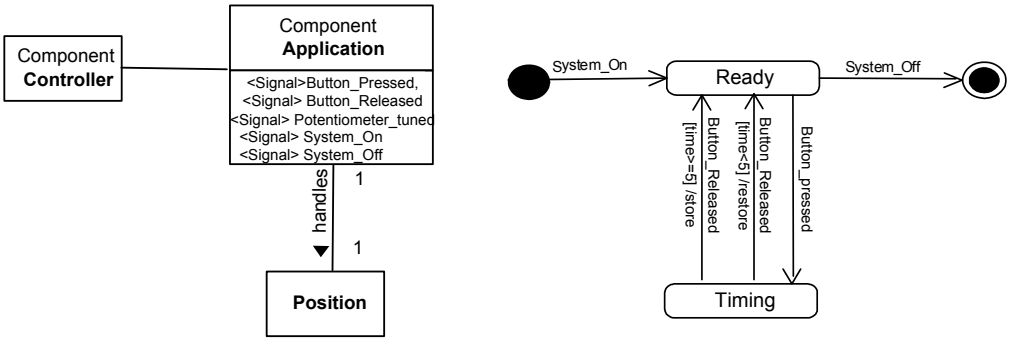


Fig. 9 UML representation of hardware



(a) Application specification - class diagram

(b) Application specification - statechart

Fig. 10 Specifications for the Application component

up of" finer-grained components, in a recursive manner, down to the level of small, primitive components.

4.3 Component specification

Since the hardware environment is pre-defined, this study focuses on the development of the software part, namely the *Driver* and the *Application* components. The Controller component is a container without any software functionality which contains the *Application* component. Figure 10(a) shows the specification level class diagram of the *Application* component in its context. *Application* 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 component (see Figure 10(b)) shows that it is in the initial state in the beginning and transits to *ready* state after the *system_on* event. When the *button_pressed* event happens, it 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 *restore*(*recall*) action or *store* action and transits back to the *ready* state.

4.4 Component realization

The specification behavior of *Application* can be internally realized through a successive refinement process; for example, it can be realized using the *Driver* component which can be again refined into several subcomponents. Figure 11(a) shows the Driver component, used to realize the functionality of the *Application* component. As shown in Figure 11(b) the realization of the *Application* component is defined in an interaction diagram specifying the interaction behavior between the refined component *Application* and the refining component *Driver*; the *button_pressed* (*button_released*) event from the user initiates the *timer_starts* (*timer_stops*) action in the *Driver*, and then, depending on the duration of the event, the *Application* component interacts with the *Driver* to either restore (recall) or store the mirror position.

Note that the *Driver* component becomes the target abstract component at the next iteration (Figure 12); the external behavior for the *Driver* may be defined in statechart in the specification process and the internal interaction behavior among *Driver* and its six refining components may be defined in sequence diagrams in the realization process.

4.5 Component Implementation

Iteratively devising specifications and realizations is continued until an existing component is found, or, until it can be implemented (no reuse). Coming to a concrete implementation from the models requires the level of abstraction to be reduced in the descriptions. First, the containment hierarchy is simplified according to the technical restrictions of the used implementation technology, i.e., through refining the containment hierarchy and mapping it to a UML model with the source code structure of the resulting system. Second, the models are

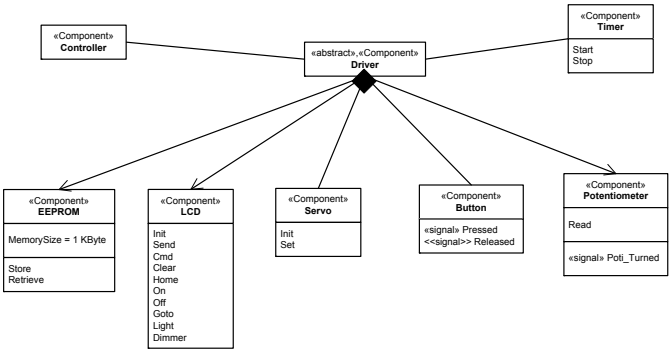
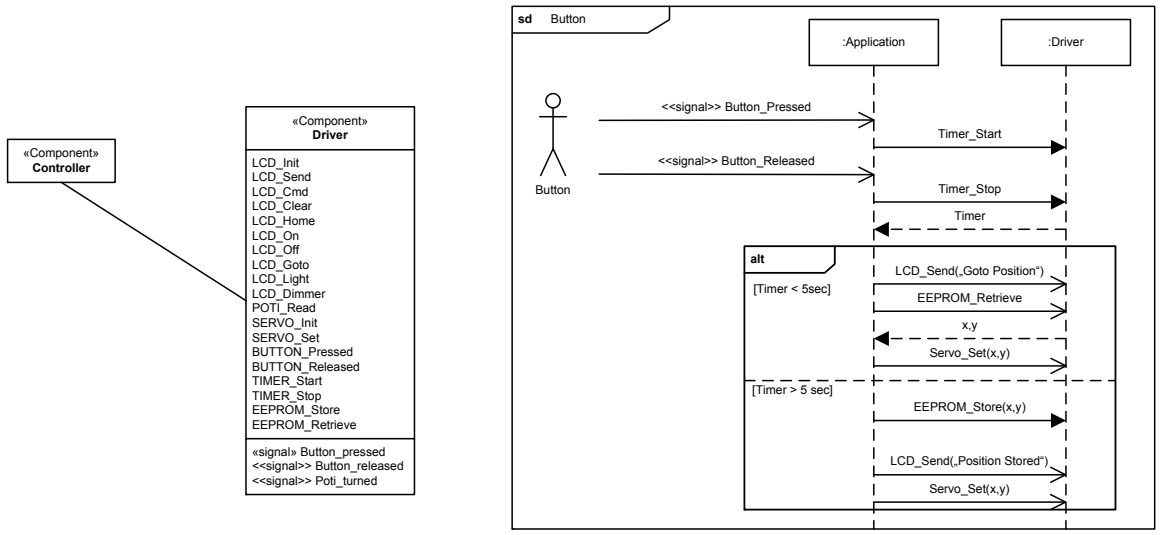


Fig. 12 Realization of the Driver component



(a) Structural realization of Application component

(b) Behavioral realization of Application component

Fig. 11 Realization of the Application component

```
#include "servo.h"

volatile int servo1, servo2;

SIGNAL(SIG_OUTPUT_COMPARE1A)
{
 //End Servo Pulse 1
 PORTB |= (0<<DDB1);
}

SIGNAL(SIG_OUTPUT_COMPARE1B)
{
 //End Servo Pulse 2
 PORTB &= ~(1<<DDB2);
}

SIGNAL(SIG_OVERFLOW1)
{
 // Trigger every 16ms
 int srv;

 PORTB |= (1 << DDB1);
 PORTB |= (1 << DDB2);

//Servo1
 srv = servo1 + 6000;
 OCR1AH = srv >> 8;
 OCR1AL = srv & 0xFF;

 //Servo 2
 srv = servo2+6000;
 OCR1BH = srv >> 8;
 OCR1BL = srv & 0xFF;

 // Reset Counter
 TCNT1H = 0x00;
 TCNT1L = 0x00;
}
```

```
void Init_Servos()
{
DDRB |= (1 << DDB1);
DDRB |= (1 << DDB2);
PORTB &= ~(0<<DDB1);
PORTB &= ~(1<<DDB2);
TCCR1A = 0x00;
TCCR1B |= (1<<CS10);
TCNT1H = 0x00;
TCNT1L = 0x00;
servo1 = servo2 = 0;
TIMSK |= ((1 << OCIE1A) | (1
OCIE1B) | (1 << TOIE1));
TIMSK |= ((1 << OCIE1B) | (1
TOIE1));
sei();
}

void Set_Servo(unsigned char
servo, unsigned int angle)
{
 int intermediate;

 intermediate = ((angle * 4)
2000);
 if (intermediate < -2000)
   intermediate = -2000;
 if (intermediate > 2000)
   intermediate = 2000;
 if (servo == 1)
   servo1 = intermediate;
 else
   servo2 = intermediate;
}
```

Fig. 13 Source code for the *servo* component.

mapped to source code, either through a code generator, or through manual mapping according to mapping guidelines as published in [34]. Figure 13 shows an example code snippet for a servo component.

5 Design Verification Approach

One major problem in component-based software development is to ensure the correctness and consistency of a system's behavior. This is not a trivial problem since behavioral complexity is high due to the composition of dozens, possibly hundreds, of components into one system. The MARMOT verification approach uses a divide-and-conquer strategy, enabled by the notion of abstract components, to address this issue.

Figure 14 illustrates the schematic view of the MARMOT design verification approach using abstract components. At the i_{th} decomposition/refinement step, it is assumed that the external structure and behavior of the i_{th} abstract component is already specified. The next step then is to realize the abstract component by specifying its internal structure and behavior. During this step it might be necessary to acquire services provided by other components, resulting in the decomposition/refinement of the abstract component. In this process, it is important to assure the behavioral consistency between the realization behavior of the i_{th} abstract component (the service user) and the specification behavior of the $(i{+}1)_{th}$ abstract component (the service provider). In this sense, the interaction consistency between refined(i_{th}) and refining components($(i{+}1)_{th}$) is defined in terms of system consistency with respect to its environment.

C1. *A system (or a refining component) is consistent with its environment (refined component) in its behavior if it either terminates normally or runs infinitely under the infinite sequence of stimuli generated from its environment (refined component).*

C2. *A system (or a refining component) is inconsistent with its environment (refined component) in its behavior if it terminates abnormally under the infinite*

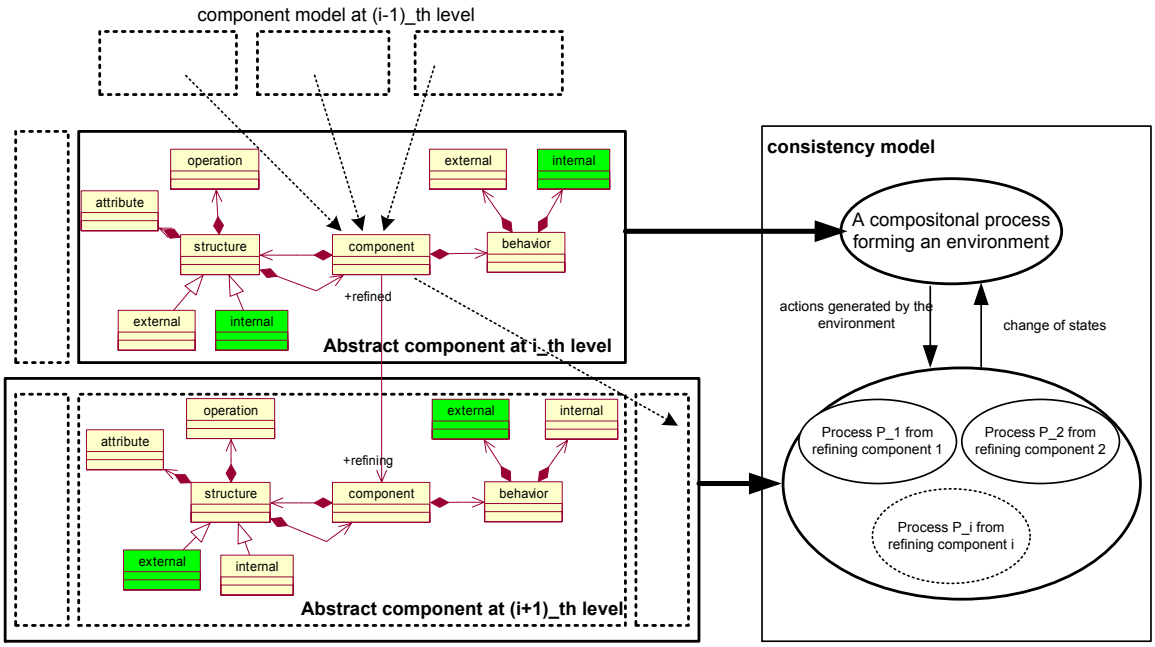


Fig. 14 A general framework for design verification

sequence of stimuli generated from its environment (refined component).

Behavioral inconsistency of the C2 type can easily be derived from behavioral inconsistency of the C1 type. This can be done by transforming the consistency issue into the problem of finding an abnormal termination for a system, composed of refining components under the environment explicitly specified by a refined component at each i_{th} refinement step. Please note that at the first-level of an abstract component, the specification of the abstract component is considered as a stand-alone system with the actual operational environment being the environment of the abstract component.

As illustrated in Figure 14, the activity of checking behavioral consistency can be systematically integrated into the development process: At any iteration of component decomposion/refinements, a modeler can choose a subject abstract-component that can be analyzed for its interaction consistency. The internal behavior of this component is then transformed into a (compositional) flow of activities, defining the environment in the consistency model. The internal structure of the subject component is used to identify refining components, to extract their external behavior, and to extract and convert their behavior into compositional reactive processes. Communication between the environment and the reactive process is based on actions of service calls (from environment to the reactive system) and the change of the state of the reactive system.

In general, the consistency model can be automatically constructed for each iteration and model-checked to ensure behavioral consistency. Figure 15 illustrates

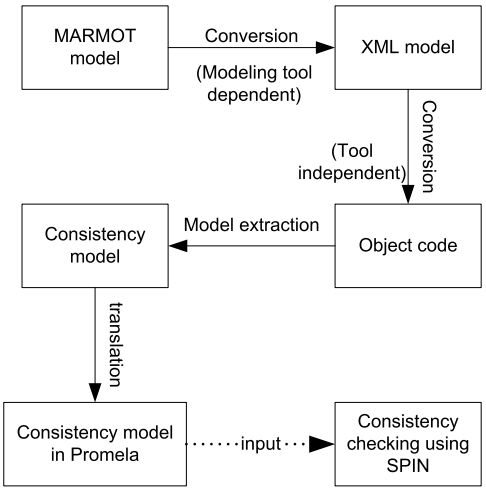


Fig. 15 A prototype realization of the general framework

the proposed prototype implementation for the construction of the consistency model. First, a MARMOT model, specified in UML, is converted into a model using XML. This model is again converted into object code following the abstract syntax of UML. This allows the verification framework to act independently of UML modeling-tools and their communication standards. Once the tool independent specification exists, the internal behavior specification(s) of the subject component and the external behavior specifications of the refining components are extracted from the object code and are transformed into the target specification language PROMELA of the model checking tool SPIN [27].

6 Formalism

This section introduces the underlying formalism that supports the design verification approach introduced in Section 5. First, the semantics of an abstract component and its externally visible behavior is defined by using the π-calculus [39], with an emphasis on communication behavior[4]. Second, the definition of the inter-relations between an abstract component and its refining components is followed by the definition of a consistency model and the notion of interaction consistency.

6.1 Specification of Abstract Components

In order to understand the behavior of a component, an abstract component is first defined using the π-calculus notation [39] as a composition of two parallel processes consisting of an interface I and an externally visible body $Spec$:

$$Comp\_spec(i, o, o\_set, g\_set, a\_set) = \\ new\ u, v(I(i, o, u, v) \mid Spec(u, v, o\_set, g\_set, a\_set))$$

Each $Comp\_spec$ has pre-defined input/output channels (i, o), a set of operations (triggering events) o_set, a set of guarding conditions g_set, and a set of actions a_set that are visible from outside the component. Thus, these are used by the component to interact with its external environment. The interfaces I and $Spec$ interchange messages and events through the internal channels u, v. Here, $new\ u, v$ represents the fact that channels u, v are dynamically created within the component with a limited scope. The symbol " | " is used to represent the parallel composition of two processes.

An interface 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. Following the π-calculus:

$$I(i, o, u, v) = i?x.u!x.I(i, o, u, v) + v?y.o!y.I(i, o, u, v)$$

Here, the symbol $i?x$, $u!x$ represents an input event on channel i, an output event on channel u, with message/signal x, respectively. Two events that are concatenated with a "." symbol occur sequentially; $i?x.u!x$ means that an input event is followed by an output event. The + 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 and the interface I connects external channels to internal channels depending on the choice of communication behavior of the component.

[4] We decided to use the π-calculus instead of formalizing by using a transition system since our intension is to emphasize external communication behavior rather than internal state transitions. However, we do use a transition system to define consistency model for verification purpose.

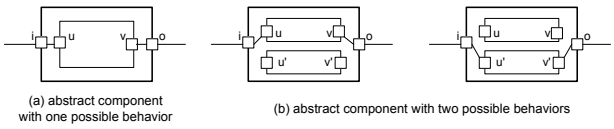


Fig. 16 The role of the interface

The purpose of defining an interface as an independent process is two-fold: First, more than one possible behavior can be specified in the abstract component. As illustrated in Figure 16, the connection between external channels i, o and internal channels u, v can be static when only one internal behavior is specified, or dynamically change depending on the choice of its behavior. Second, the communication behavior can change independently from functional behavior. For example, more detailed communication behavior, such as a message passing mechanism, synchronization, and buffering scheme, can be specified in I to test various communication methods [12].

The process $Spec$ is a sequential process defined with a series of process states from $Spec_0$ (i.e., $Spec = Spec_0$). It represents the process at an initial state, whereby $Spec_i$ represents the process at the i_{th} state. In detail:

$$Spec_i(u, v, o\_set, g\_set, a\_set) = \\ \sum_{op \in o\_set} u?x.[x = op] Action\_spec^i_{op}(u, v, o\_set, g\_set, a\_set) \\ +u?x. \prod_{op \in o\_set} [x \neq op] Spec_i(u, v, o\_set, g\_set, a\_set) \\ Action\_spec^i_{op}(u, v, o\_set, g\_set, a\_set) = \\ \sum_{g \in g\_set} [g]v!f(i, op, g).Spec_j(u, v, o\_set, g\_set, a\_set)$$

$Spec_i$ receives a message x, checks whether it matches with one of the operations in o_set, and performs corresponding actions $Action\_spec^i_{op}$ if it matches, and does nothing if it does not match with any of the operations in the set. Here, the notation $\prod_{op \in o\_set}[x \neq op]$ is an abbreviation for $[x \neq op_1][x \neq op_2] \ldots [x \neq op_n]$ for all $op_i \in o\_set$. $Action\_spec^i_{op}$ defines a series of actions that need to be performed by the component for a particular operation. The actions, based on process state, operation, and guard condition, are notified to the internal output channel v. This channel is forwarded to the external output channel o by interface I. Finally, the process reduces to $Spec_j(u, v, o\_set, g\_set, a\_set)$ whereby the mapping from $Spec_i$ to $Spec_j$ is pre-defined by conditions on message values. In other words, there is a mapping from $\{(i, x) \mid$ *process state* i, *message value* $x\}$ to $\{j \mid$ *process state* $j\}$. This mapping can be extracted from the state diagrams in the MARMOT specification model.

For example, the statechart in Figure 10(b) describes the specification of an abstract component. In this case, the *Spec* process can be specified as follows;

$$\begin{aligned}
Spec_0(...) &= u?x.[x = System_On]Spec_1(...) \\
&\quad + u?x.[x \neq System_On]Spec_0(...) \\
Spec_1(...) &= u?x.[x = Button_pressed]Spec_2(...) \\
&\quad + u?x.[x = System_Off]Spec_3(...) \\
&\quad + u?x.[x \neq Button_pressed][x \neq System_Off] \\
&\quad\ Spec_1(...) \\
Spec_2(...) &= u?x.[x = Button_Released]Action_spec_2^{BR}(...) \\
&\quad + u?x.[x \neq Button_Released]Spec_2(...),
\end{aligned}$$

where

$$\begin{aligned}
Action_spec_2^{BR}(...) &= [time < 5]v!restore.Spec_1(...) \\
&\quad + [time >= 5]v!store.Spec_1(...)
\end{aligned}$$

The *Spec* process can also be modeled as a labeled transition system (LTS) with final states.

Definition 1 A labeled transition system (LTS) *is a quintuple (S, L, R, I, T), where S is a set of states, L is a set of labels (actions) of the system, $R \subseteq S \times L \times S$ is a transition relation, I is a set of initial states, and T is a set of terminal states.*

A process $P = (S, L, R, I, T)$ transits from a state s_i to a state s_{i+1} if and only if P is in state s_i and there exists $l_i \in L$ such that $(s_i, l_i, s_{i+1}) \in R$, briefly denoted as $P \xrightarrow{l_i} P'$. The notation $P(s_i)$ is used to denote that the process P is in state s_i. Note that equivalently the state transition can be expressed as $P(s_i) = \sum_{k=1}^{n} l_{ik}.P(s_{i+k})$ in π-calculus (assuming that there are n possible transitions from state s_i).

Composition of two processes I and *Spec* is denoted by $I|Spec$ representing the parallel composition of I and *Spec* with synchronization of actions common to both of their labels and interleaving of the others. The transition of a process sometimes generates actions either internally or externally. Such a transition is denoted with $(s_i, l_i/l_i', s_{i+1})$ or $s_i \xrightarrow{l_i/l_i'} s_{i+1}$ meaning that the process transits from s_i to s_{i+1} triggered by an action l_i generating an action l_i'. In this sense, the set of labels L can be classified into two sets: A set of actions L_t that triggers transitions in R, and a set of actions L_g generated from transitions in R. L_t and L_g are not necessarily disjoint.

6.2 Realization of Abstract Components

The specification of a component defines the externally visible behavior of the component regardless of how it is realized internally. The externally visible behavior can be uniquely defined based on the functionality that is provided by the component. However, each functionality can be realized in many different ways. The MARMOT realization activity defines a special approach towards the realization of a component specification via decomposition and refinement. The focus is on making this activity as flexible as possible so that changing a certain realization of a component does not affect the behavior of the overall system.

In the MARMOT realization process the *Spec* process is refined with the *Real* process, as defined below, consisting of a number of parallel *SubComp* processes that collaboratively realize it. Note that each *SubComp* is considered as an independent component in its own right. Thus, it can be recursively specified as an abstract component in the same way as *Comp_Spec*.

$$\begin{aligned}
&Real(u, v, o_set, g_set, a_set) = new\ \{u_k, v_k\}_k(\\
&\quad !_k\ SubComp(u_k, v_k, sub_o_set_k, sub_g_set_k, sub_a_set_k) \\
&\quad |\ Comp_Real(u, v, \{u_k, v_k\}_k, o_set, g_set, a_set))
\end{aligned}$$

The *Real* process defines the communication channels among an abstract component and its subcomponents, whereby the *Comp_Real* process defines refined behavior of the abstract component. Here, the notation $\{u_k, v_k\}_k$ is used to abbreviate k pairs of input/output channels. $!_k$ is used to abbreviate the parallel composition of k *SubComp* processes whose interrelation is defined in a function f that maps each pair of channel and message to a sequence of actions depending on guarding conditions:

$$\begin{aligned}
&f : \{(w, op, g) \mid w \in c_set, op \in o_set, g \in g_set\} \\
&\quad \longrightarrow\ \{< (w_i, a_i) >_i \mid w_i \in c_set, a_i \in a_set\}
\end{aligned}$$

$$\begin{aligned}
&Comp_Real_i(c_set, o_set, g_set, a_set) = \\
&\qquad \sum_{w \in c_set, op \in o_set, g \in g_set} w?x.[x = op]f(w, op, g)Comp_Real_j(...) \\
&\qquad + w?x.\prod [x \neq op]Comp_Real_i(...)
\end{aligned}$$

c_set represents the set of internal input/output channel pairs $\{u, v, \{u_k, v_k\}_k\}$. Note that the function f is used to wire sub-components and can be changed independently from the implementation of each *SubComp*, supporting a flexible design for the component-based development of software systems.

For example, the class diagram in Figure 11(a) shows the structural realization of the abstract component *Application* which is refined with the *Driver* component. In this case, the *Real* process for the *Application* component has a pair of internal input/output channels, e.g., {u_driver, v_driver}, for communicating with the *Driver* component, in addition to

the input/output channels $\{u, v\}$ defined in the specification level. Since there is only one refining component the following condition holds: If a message arrives at u, it delivers the message to the destination u_driver. f is defined as $f(u, button\_pressed, true) = \{< u\_driver, Timer\_Start >\}$ assuming that the realization behavior of the *Application* is specified so that the message $Timer\_Start$ is to be delivered to the *Driver* component when *button_pressed* event occurs.

The refined behavior of the *Application* component is specified by a series of sequence diagrams. Figure 11(b) illustrates one of these sequence diagrams that specifies the interaction behavior between the refined component *Application* and the refining component *Driver* when a button-related signal is received. Note that the sequence diagram is used to specify detailed actions taken by the *Application* for each external signal. The following is a fragment of the *Real* process derived from the sequence diagram;

$$Real(...) = new\{..u\_driver, v\_driver..\}(\\ \quad Driver(u\_driver, v\_driver, ..) \mid Comp\_Real(...))\\ Comp\_Real_1(...) = u?x.[x = button\_pressed]\\ \quad u\_driver!Timer\_Start.Comp\_Real_2(...) + \ ...$$

6.3 Consistency Model

As briefly introduced in Section 5, the internal behavior is specified by an abstract component as the environment of the set of refining abstract components that are used to realize the abstract component, where the compositional process of refining abstract components is considered as a stand-alone system. In this sense, a more restricted form of composition is considered in the following: A process $P = (S^p, L^p, R^p, I^p, T^p)$ may be restricted by its environment $E = (S^e, L^e, R^e, I^e, T^e)$, denoted by $P \uparrow E = (S^p \times S^e, L^p \cup L^e, R^p \times R^e, I^p \times I^e, T^p \times T^e)$, meaning that the environment E generates, and, thus, constrains the (sequence of) actions that triggers transitions in P.

a. $$\frac{s_i^p \xrightarrow{l_i^p} s_{i+1}^p, \quad s_i^e \xrightarrow{l_i^e / l_i^{e\prime}} s_{i+1}^e}{(s_i^p, s_i^e) \xrightarrow{l_i^p, l_i^e / l_i^{e\prime}} (s^p{}_{i+1}, s^e{}_{i+1})} \ (l_i^{e\prime} = l_i^p)$$

b. $$\frac{s_i^p \xrightarrow{l_i^p} s_{i+1}^p, \quad s_i^e \xrightarrow{l_i^e / l_i^{e\prime}} s_{i+1}^e}{(s_i^p, s_i^e) \xrightarrow{\{\}, l_i^e / l_i^{e\prime}} (s_i^p, s_{i+1}^e)} \ (l_i^{e\prime} \neq l_i^p)$$

Rule a says that both P and E follow transitions when the triggering action for a transition of P is generated from a transition of the environment E. Rule b says that P stays in the same state when E does not generate an action that triggers a transition in P.

The consistency model comprises the behavioral description of refining components and the realization of each service of a refined component. Both can be considered as a parallel composition of processes.

Definition 2 *A consistency model for the i_{th} refinement is a closed system $P \uparrow E$, with $P = P_1|P_2|\dots P_n$ being a compositional process consisting of the externally visible behavioral specification of each refining component, and $E = E_1|E_2|\dots E_m$ being a compositional process consisting of the realization behavior of each service provided by the refined component.*

Note that the notion of environment is a relative concept. At each i_{th} refinement step, a set of refining components is considered to be a stand-alone system whose behavior is restricted by the internal behavior specified in the refined component acting as an environment of the system.

The consistency model is used to check interaction consistency in the MARMOT component refinement activity with respect to *termination* and *progressiveness*.

Definition 3 Termination: *A process P terminates normally (in state s) under environment E, denoted as $Terminate(P(s)) \uparrow E$, if and only if P terminates to a state s that belongs to the pre-defined set of terminal states T, i.e., $P(s) \wedge s \in T$, and there is no $l \in L, s' \in S$ such that $(s \xrightarrow{l} s') \uparrow E$.*

- *A compositional process $P = P_1|P_2|\dots P_n$ terminates normally, if each of its sub-processes P_i with a non-empty set of terminal state T_i terminates normally in state s_i under environment E_i. Here, E_i is a parallel composition of the environment of P and all the processes P_j with $j \neq i$, i.e., $Terminate(P_i(s_i)) \uparrow E_i$ where $E_i = E|P_1|P_2\dots P_{i-1}|P_{i+1}|\dots P_n$.*

Definition 4 Progressiveness: *A process P is progressive (in state s) under environment E, denoted by $Progress(P(s)) \uparrow E$, if and only if there is a sequence of states $s_1, s_2, \dots, s_n \in S$ and a sequence of labels $l_1, l_2, \dots, l_{n-1} \in L$ such that $s_n \neq s$ and $(s \xrightarrow{l_1} s_1 \dots \xrightarrow{l_{n-1}} s_n) \uparrow E$.*

- *A compositional process P is progressive under an environment E, if and only if there exists a sub-process P_i which is progressive under its environment $E_i = E|P_1|P_2\dots P_{i-1}|P_{i+1}|\dots P_n$.*

Definition 5 Interaction consistency: *A compositional process P is consistent with its environment in a state $s_k = (s_{k_1}, s_{k_2}, \dots, s_{k_n})$ after k_{th} transition, denoted by $Consistent(P(s_k)) \uparrow E$ if and only if $Terminate(P(s_k)) \uparrow E \vee Progress(P(s_k)) \uparrow E$.*

Simply speaking, a process is consistent, at a specific execution time, if and only if it is normally terminated or progressive.

Interaction consistency can be checked by applying formal verification methods and automation tools, such as CSP/FDR [36], theorem proving [45], and model checking [14,27]. MARMOT uses the model checker SPIN because SPIN has a built-in invalid end-state verification option which corresponds to the notion of interaction consistency.

6.4 From Abstract Components to PROMELA

The goal is to incorporate automated checking mechanism into the MARMOT development process so that a system can be formally checked at the earliest development steps. To this end, the proposed verification framework integrates the model checker SPIN [27] as a back-end verifier for the MARMOT concept of abstract components.

The use of SPIN requires a translation of MARMOT models into PROMELA [26], the input language of SPIN. The syntactic transformation from MARMOT into PROMELA is based on the formal meaning of abstract components as defined in the previous sections.

Figure 17 shows the skeleton of the syntactic translation from MARMOT (UML based) models into 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 $Spec_i$. 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.

As explained in Section 4, UML diagrams, such as statecharts for specification behavior and sequence diagrams for realization behavior, are used to specify MARMOT abstract components. The proposed prototype translation converts sequence diagrams into statecharts [51] and interprets statecharts with LTS semantics. The translation of these statecharts into PROMELA follows the rules described in Figure 17 and the LTS-to-PROMELA translation proposed in [38].

7 Design Verification of the Mirror Control System

This section presents the application of the design verification approach, proposed in Section 5 and Section 6, to the case example introduced in Section 4. Please note that all the PROMELA codes presented in this section are edited for better readability[5].

[5] Our prototype translation tool uses encodings of state variables, and, thus, its PROMELA code is rather lengthy.

7.1 Translation of component specification

The process *Comp_Spec* for the Application component is specified in PROMELA 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);
}
```

Here, *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*. The input, output channels i, o are declared in the signature of the component. Inside the process *Comp_spec*, u, v are declared as channels with the message type *mtype*. Furthermore, these 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 sub-processes using the keyword *run*.

The interface part is directly translated into PROMELA proctype that simply delivers messages to/from internal input/output channels as described below:

```
proctype Interface(chan i,o,u,v){
  mtype x;
  do
  ::i?[x] -> i?x; u!x;
  ::v?[x] -> v?x; printf("%d", x);
  od;
}
```

Here, the statement $i?[x]$ becomes true if a message is at the channel i. Note that the output messages from the internal output channel v is printed, instead of being delivered to the output channel o, to make it easy to read the output result.

The behavior of the *Spec* process is derived from the statechart of the *Application* component described in Figure 10(b) as its translated model is shown in Figure 18: 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 *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 stays in *Spec_2* (line 23). Note that predicate abstraction [23] 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*; first, $time < 5$ is replaced with

	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] → goto $state_i$; fi;

Fig. 17 Syntactic translation from abstract component to PROMELA [13]

```
 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;
     }
```

Fig. 18 Spec process in PROMELA

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 a non-deterministic choice at this abstract level, the guarded action is replaced with a "non-deterministic choice between *store* and *restore*" as expressed in line 19 – line 22.

```
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;
}
```

Fig. 19 Environment model of *Application* in PROMELA

7.2 Verification of the Component Specification

Once the abstract component has been formally specified, the behavior of the abstract component can be checked to see if it is consistent with its environment. Note that the environment of the *Application* component is specified in the use case scenarios given in Figure 8, which can be directly transformed into PROMELA as shown in Figure 19. Here the statements enclosed by *do..od* are repeated indefinitely as the *poti_tuned* signal or the *button_pressed* signal followed by the *button_released* signal is generated non-deterministically.

This environment process *env* is composed with the *Comp_Spec* process producing a closed consistency model. This consistency model is fed into the SPIN model checker for automated consistency checking. After exhaustive examination for process deadlock situations, SPIN generates a counter example (Figure 20) showing an execution trace that can reach a process deadlock. If there is a sequence of signals from *Potentiometer* that occupies the input channel of the $Interface$ process, the

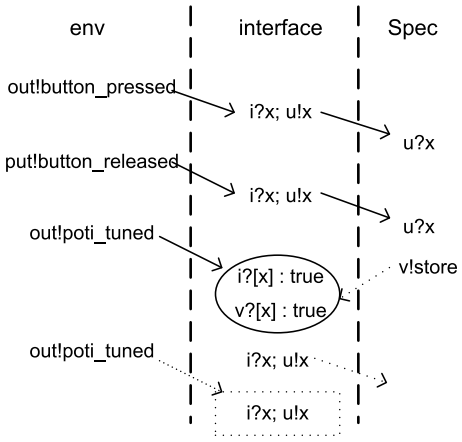


Fig. 20 A counter example trace

handling of the internal output message can be postponed indefinitely, making the system stall.

Figure 20 illustrates the last part of the counter example; *env* generates *button_pressed* signal followed by *button_released* signal. Both signals are delivered to the *Spec* process through the $interface$ sequentially, making the *Spec* process transits to *Spec_2* state(line 16 in Figure 18) where it tries to send *store* message to the internal output channel v. At the same time, however, the *env* process generates *poti_tuned* signal. Since the $interface$ can handle only one of the two events $i?x$ and $v?x$, it has to serialize the handling of the two events. If it chooses to handle $i?x$ first, then it generates signal $u!x$ which cannot be received by the *Spec* process because the process is still trying on the unfinished action $v!store$. In the mean time, the *env* again generates *poti_tuned* signal which cannot be handled by $interface$ which stalls on generating signal $u!x$. No further progress is possible in this situation.

After careful review of the identified counter examples, the source of this problem was found as the message handling mechanism of the interface. This problem can be addressed by two alternative design choices: The first possible choice is to change the behavior of the interface to loose additional messages instead of blocking them until they are handled. The second possible choice is to refine the $Interface$ process with two independent message handlers that handle input messages and output messages independently.

$$I(i,o,u,v) = I_{in}(i,u) \mid I_{out}(o,v)$$
$$I_{in}(i,u) = i?x.u!x.I_{in}(i,u)$$
$$I_{out}(v,o) = v?x.o!x.I_{out}(v,o)$$

This is translated into the following PROMELA code,

```
proctype Interface(chan i,o,u,v){
  run input_handler(i,u);
  run output_handler(o,v);
}
```

```
proctype input_handler(chan i,u){
  mtype x;
  do
  ::i?[x] -> i?x; u!x;
  od;
}
proctype output_handler(chan o,v){
  mtype x;
  do
  ::v?[x] -> v?x; printf("%d", x);
  od;
}
```

The second option was chosen in this case study because there is a possibility of loosing important messages by following the first option. After refining the message handler following option two, SPIN verifies that the component specification is consistent with its environment. The comprehensive search for consistency checking runs through 12,799 states and 31,412 transitions consuming 3.6 M bytes of system memory and 0.019 seconds. Verification was performed on a PC with 2G Hertz Pentium II processor and 2G bytes of memory.

7.3 Translation of Component Realizations

The *Application* component is realized using the services provided by the *Driver* component as partly illustrated in Figure 11(b). 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. In this realization model all sub-components are designed as passive objects with the *Application* component (the container of the driver components) acting as an active signal 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.

Figure 21 shows part of the realization model of the *Application* component in PROMELA. The figure provides only the part changed from the *Spec* process (Figure 18), reflecting the realization behavior specified in Figure 11(b). The *Real* process includes two internal communication channels *di* and *d_o* to/from the refining component *Driver*. For each event, *Real* specifies more refined actions taken by the *Application* component. For example, the *Real* process sends $Timer\_Start$ message to the input channel of the *Driver* component and transits to *Spec_2* state (line 15–16 in Figure 21) when *button_pressed* event arrives, whereas the *Spec* process just transits to *Spec_2* for the same event (line 12 in Figure 18).

It was verified using SPIN that this realization of the *Application* component is consistent with its environment by replacing the *Spec* process with *Real* process in PROMELA. it comprehensively searches through 7,375 states and 16,557 transitions using 33.4 M bytes of sys-

```
1: proctype Real(chan u,v){
2:    mtype x;
  /* create internal channel for driver */
3:   chan di = [1] of {mtype};
4:   chan d_o =[1] of {mtype};
5:   run Driver(di, d_o);

6:  Spec_0:
     ....

12: Spec_1:
13:  u?x ;
14:  if
15:  :: x==button_pressed ->
            di!Timer_Start;  goto Spec_2;
16:  :: x==system_off -> goto end_state;
17:  :: else -> goto Spec_1;
18:  fi;

19: Spec_2:
20:   u?x;
21:   if
22:   :: x==button_released ->
23:        di!Timer_Stop;
24:        d_o?x;
25:        if
26:        :: 1 -> atomic{ di!LCD_Send;
                      di!EEPROM_Retrieve;
                      d_o?x; di!Servo_Set; }
                   goto Spec_1;
27:        :: 1 -> atomic{ di!EEPROM_Store;
                   di!LCD_Send; di!Servo_Set;  }
                   goto Spec_1;
28:        fi;
29:   :: else -> goto Spec_2;
30:   fi;
31: end_state: goto Spec_0;
}
```

Fig. 21 Application realization in Promela

```
1: proctype Driver_Real(chan di, d_o){
2:  chan bi = [1] of {mtype};
3:  chan bo = [1] of {mtype};
4:  chan li = [1] of {mtype};
5:  chan lo = [1] of {mtype};
6:  chan pi = [1] of {mtype};
7:  chan po = [1] of {mtype};
8:  chan ei =[1] of {mtype};
9:  chan eo = [1] of {mtype};
10: chan si = [1] of {mtype};
11: chan so = [1] of {mtype};
12: chan tin = [1] of {mtype};
13: chan tout = [1] of {mtype};
  /* run each driver component */
14: run Button_driver(bi,bo);
15: run Servo(si,so);
16: run LCD(li,lo);
17: run Potentio_driver(pi,po);
18: run EEPROM(ei, eo);
19: run Timer(tin, tout);
20: mtype x;
21: do
22: ::di?[x] -> di?x;
23:      if
24:      ::x==EEPROM_Retrieve -> ei!x;
                          eo?x; d_o!x;
25:      ::x==LCD_Send -> li!x;
26:      ::x==Servo_Set -> si!x;
27:      ::x==Timer_Stop -> tin!x;
                       tout?x; d_o!x;
28:      ::else ->skip;
29:      fi;
30: od;
}
}
```

Fig. 22 Comp_Real process in PROMELA

tem memory and 0.01 seconds concluding that this model is consistent in its interaction behavior.

7.4 Checking Interaction Consistency

So far, this study has shown how the *Application* abstract component was specified and refined by the *Driver* component in the first iteration. The next step is to specify the *Driver* component and realize it through decomposition. For example, the *Driver* abstract component is refined into six subcomponents at the second refinement iteration as shown in Figure 12. Figure 22 shows the translated version of the realization behavior and structure of the *Driver* component. The *Driver_Real* process contains six pairs of internal input/output channels connected to its sub-components. It activates its subcomponents and deliver input messages to corresponding sub-components.

Note that this *Driver_Real* process is a refinement of the *Driver* process activated in *Real* (line 5 in Figure 21), and, thus, interaction consistency can be verified by simply replacing *Driver* with *Driver_Real* (and adding the specification behavior of each subcomponent). This can be a straightforward way of refinement checking, but it also increases verification complexity as refinements add (and never remove) components. Therefore, this study takes the *Driver_Real* as an execution environment of the 6 subcomponents whose behavior is constrained by the *Real* process but independent from any higher level processes related to the *Application* component.

Figure 23 illustrates the consistency model derived from the refinement process of the *Application* component. In this example, checking interaction consistency by simply replacing the *Driver* with *Driver_Real* (Figure 23 (c)) requires SPIN to search through 128,004 states and 187,731 transitions, consuming 45 M bytes of system memory and 0.2 seconds. On the other hand, the iterative consistency checking (Figure 23 (b)) searches through 4,577 states and 13,929 transitions, consuming 33.4 M bytes of system memory and 0.01 seconds. Note that checking consistency model 3 in this way does not need more resources than checking consistency model 1 or 2, whereas checking direct composition consumes more time and memory. The resource consumption for checking direct composition increases exponentially since refinements successively add more components.

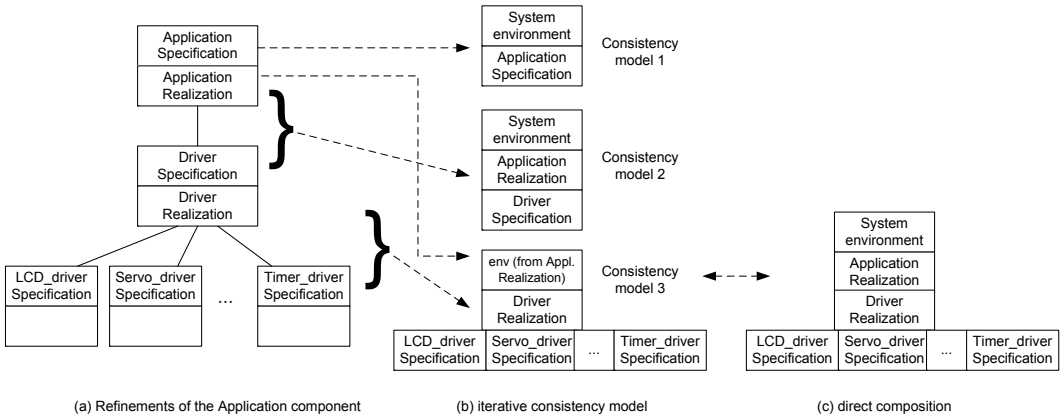


Fig. 23 Iterative consistency models in the refinement process

8 Application to a larger System

To obtain more evidence regarding the effects of the proposed design verification approach it has been applied in a recent project for reverse engineering TinyOS, from its code written in nesC [22] to application level abstract components, in a bottom-up manner. The motivation for this project was to provide a basis for model-based development for existing embedded software by systematically extracting high-level abstract components from program code. The reverse-engineered abstract components become a basis for the model-driven development of the same application domain.

TinyOS [31] is an open-source operating system for wireless sensor networks developed at the University of California at Berkeley. Its core code consists of less than 4000 bytes of code, which consumes less than 256 bytes of data memory. It supports event-based multi-tasking, aiming at low-cost embedded operating systems suitable for embedded networking. The TinyOS code includes about 60 files for defining interfaces, 52 files for defining system components, about 129 library files, and 66 files for a hardware platform for defining platform-specific components. TinyOS supports 12 different hardware platforms. Among these, the code for *hmote*2420 was chosen because that specific hardware was available in the authors' research group.

Currently, 60 interfaces, 18 system components, 3 library files, and 48 platform-specific components are being reverse-engineered in abstract components, validated using model-based simulation tools in Rhapsody, and verified using the model checker Spin. Figure 24 shows a part of the component tree for the TinyOS; in the figure, $Msp430AlarmC$, $TransformCounterC$, and $TransformAlarmC$ are final components which have been physically realized. Others with *abstract component* stereotype are abstract components that are roots of their own tree of abstract components.

Each abstract component is verified for its behavioral consistency. For example, in Figure 24, $Msp430Timer32khzC$ is realized with a composition of $Msp430TimerCapComP$, $Msp430TimerCommonP$, and two instances of $Msp430TimerP$. The behavior of each of these components is translated into Promela code as an independent process, and the structural information, e.g., port binding and dependency, is used to wire those independent processes. Using Spin, the communication consistency of its realization behavior is verified within 0.45 hours, consuming 6G bytes of memory when an exhaustive search method is used. The memory consumption decreases to 1.6 G bytes but the verification time increases to 1.2 hours when compression is used.

After the verification of its realizing components, the abstract behavior of $Msp430Timer32khzC$ is again verified concerning its behavioral consistency. In this case, the detailed internal communication behavior is ignored, and the focus is rather on the behavior w.r.t. external communication. It took less than 5 minutes and 2.5 G bytes of memory to verify this consistency using exhaustive verification.

$Msp430Timer32khzC$ is again composed with $Msp430AlarmC$ to realize $Alarm32khz16C$. Each of the abstract behavior specification of the two abstract components is then translated into Promela to check the internal communication consistency. It took about 1.08 hours, consuming 15.6 G bytes of memory using exhaustive verification, and about 2.44 hours and 2.15 G bytes using compression. The external behavior of $Alarm32khz16C$ is much simpler, since all internal communication between $Msp430Timer32khzC$ and $Msp430AlarmC$ is hidden, and, thus, it took only 3.18 seconds and 1 G bytes of memory to check the communication consistency.

Table 1 summarizes the verification performance. The experiment shows that each abstract component can be reused with its verification result and that the verification complexity does not add up as the level of abstraction is lowered or increased. Performance depends

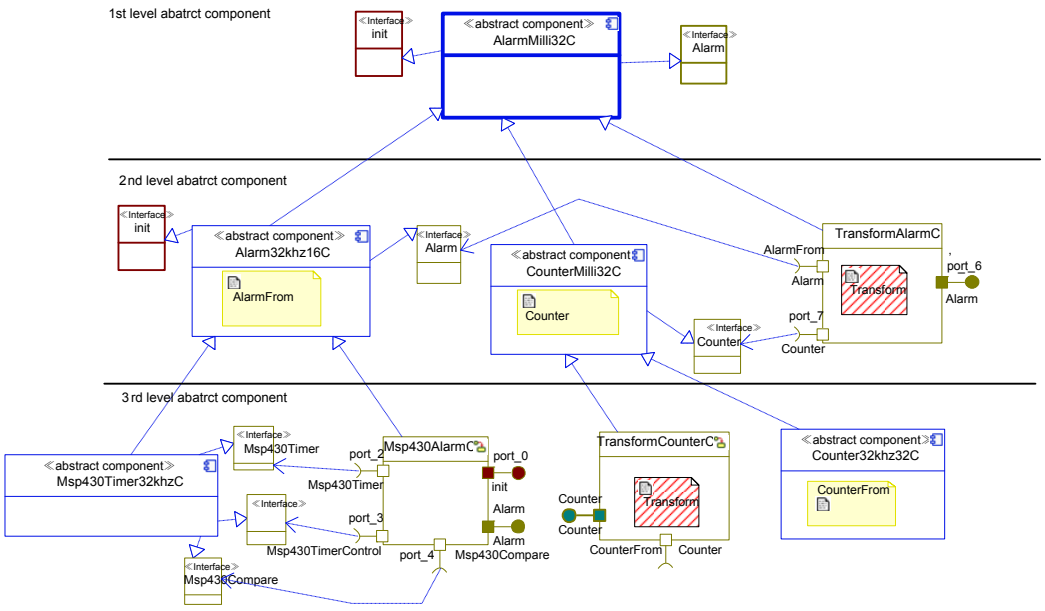


Fig. 24 A tree of abstract components

on the realization complexity at each refinement step. The experiment was performed on a Sun workstation T5240 with 1.2GHz CPU and 64G bytes of memory.

9 Discussion

Component-based software development promises higher-than-normal reuse rates and shorter time-to-market, while at the same time providing a high level of quality. Quality promises are based on the assumption that system quality will benefit from the assembly of "known-to-be-good" components. Since the future usage of components cannot be foreseen, the interaction-related behavior of components needs specific attention. Quality assurance activities such as testing or reviews with a specific focus on component interaction are a step in the right direction, but are limited in their effects when it comes to embedded systems. One solution might be the use of formal methods.

The use of formal methods in embedded system development has been an active research issue for almost a decade [33,50]. Unfortunately, formal methods are not commonly used in software industry, mainly due to the lack of experience/evidence, supporting tools, and methodologies [32]. This situation can be altered by integrating formal methods into existing development methodologies so that the application of formal methods can be seen as a routine task within the existing development process. The use of structured methodology helps making the application of formal methods more systematic by providing a gradual but seamless transition from early design to actual implementation.

The mirror-control case study has demonstrated that the application of gradual formal transformation and verification helps identify design errors early in the development cycle. The preliminary result of the design verification approach on complex embedded software looks promising; the experiment shows that design verification based on abstract components does not increase complexity exponentially as the level of abstraction is lowered or increased.

9.1 Choice of Modeling Language

Formal languages are ideal for formal development and/or formal verification approaches since they are precise, unambiguous, and can be formally checked. Some examples are requirement specification languages, architecture description languages [41,37], modeling languages for communicating processes, such as CSP and π-calculus, and languages specifically designed for formal verification, such as SMV and PROMELA. Nevertheless, these formal languages have not been widely accepted in practice mainly because; (1) they are generally difficult to learn, (2) each of these formal languages puts its focus on a certain aspect of system, and, thus, difficult to find a language suitable for the whole development process, (3) it is difficult to integrate different aspects of a system when they are specified in different formal languages, and (4) they are generally good at specifying details, but are not appropriate in providing structural information in a large view.

Due to the stated reasons, the semi-formal language UML is widely accepted despite its ambiguous semantics

name	type	search depth	states	transitions	memory (compression)	time (compression)
Msp430Timer32khzC	realization	1,494,716	2.4e+07	1.22e+08	6,068.6 (1,609)	1.64e+03 (3.93e+03)
Msp430Timer32khzC	specification	392,826	8e+06	2.92e+07	2,461 (1,062.8)	256 (627)
Alarm32khz16C	realization	9,047	4.3e+07	2.02e+08	15,621 (2,146)	3.88e+03 (8.8e+03)
Alarm32khz16C	specification	761	283,461	438,704	1,499 (821.6)	3.18 (7.8)

Table 1 Experiment data: model checking communication consistency

and potentially unclear usage of different diagram notations. The light-weight formal approach [17,30], that uses a semi-formal language for modeling and that performs formal verification by translating to a formal language, has been considered an alternative and practical solution. The proposed approach follows the same line adopting the light-weight approach using UML instead of using π-calculus or PROMELA as the modeling language.

9.2 Scalability

Automated verification techniques such as model checking suffer from the notorious problem of state-space explosion. The complexity of the model checking algorithm is typically linear to the size of the model and formula [14]. Since the size of the model (in terms of the number of states) grows exponentially as the number of state variables and the number of interleaving processes in the system increases, the technique often requires a large amount of memory and time for checking realistic models.

This study's approach tries to alleviate the state-space explosion problem by localizing the verification problem. It focuses on a consistency model extracted from realization-specification behavior at each refinement iteration. As shown in Section 7 and 8, the approach provides a systematic way of controlling the verification complexity.

9.3 Tools

9.3.1 CASE tool support

Since MARMOT uses standard UML models it can, in principle, be applied using most CASE tools. However, to relieve developers from creating standard model sets or to manually check MARMOT's built-in rules and dependencies a prototype plug-in for IBM's RationalRose toolset was developed. Currently, an additional ECLIPSE plug-in that makes use of Subclipse and UML2Tools is being developed. Watch for release announcements at *www.imenco.org*. In addition, the MEROBASE component locator (http://merobase.com/), developed at the University of Mannheim, can be used to quickly identify reusable components. The authors are currently working towards an adaptation of MEROBASE for embedded systems.

9.3.2 Translator

The proposed design verification approach can be automated independently from the modeling tool support as explained in Figure 14 and Figure 15. To demonstrate its feasibility and efficiency, a prototype translator from MARMOT abstract components to PROMELA was developed as an ECLIPSE plug-in. An initial version is available for demonstration at http://m80.knu.ac.kr/∼sselab/marmot.html.

9.3.3 Counter-example Re-play

It should be noted that the counter examples generated from the SPIN model checker may not be in a familiar form to engineers. To assist in counter-example comprehension, a tool-support is being developed for the counter-example re-play that interprets counter examples generated from the SPIN model checker, and converts them into UML sequence diagrams so that engineers can re-play it using simulation facilities provided by commercial UML case tools. An initial version of the tool is also available for demonstration at http://m80.knu.ac.kr/∼sselab/marmot.html.

9.4 Empirical Evaluation

To evaluate the MARMOT approach concerning its characteristics and benefits an empirical study in the form of a small experiment (quantitative) was performed [7] to compare the effects of MARMOT in embedded systems development to other approaches such as the Unified process and agile development (w.r.t. reuse, quality, effort, etc.) regarding its impact on reuse and quality. The results indicate that using MDD and CBD for embedded system development will have a positive impact on reuse, effort, and quality. However, similar to product-line engineering projects, CBD requires an upfront investment. Therefore, all results have to be viewed as initial. This has led to the planning of a larger controlled experiment to obtain more objective data.

9.5 Future work

This paper has demonstrated that the application of gradual formal transformation and verification helps identify design errors and ensure behavioral correctness early in the development cycle. Nevertheless, this study

is considered as a starting point requiring further investigation of several issues:

- Channels and buffers need to be formalized in more detail in order to express different communication mechanisms and functional correctness of an interaction.
- The design verification framework can be naturally extended to include the verification of functional properties, which requires specifying the properties in temporal logic. A user-friendly way of verification for functional properties needs to be developed.
- The timing issue is very important in embedded systems, but it has not been considered in the proposed approach, yet. A systematic method to divide and conquer the timing issue is to be investigated.
- The effectiveness of proposed approach needs to be evaluated on the physical implementation of the MARMOT design in order to be claimed practical. Issues related to energy consumption, timing, and utilization of limited memory [32] are especially important in embedded systems. Such issues will be investigated within the same verification framework in the future.

References

1. J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, September 2005.
2. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
3. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 2004.
4. Colin Atkinson, Joachim Bayer, and Christian Bunse et al. *Component-based Product Line Engineering with UML*. Addison-Wesley Publishing Company, 2002.
5. Andrea Bondavalli, Mario Dal Cin, Diego Latella, Istvan Majzik, Andras Pataricza, and Giancarlo Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems – Science and Engineering*, 16(5):265–275, September 2001.
6. Alan W. Braun and Kurt C. Wallnau˙The Current State of CBSE. *IEEE Software*, 1998.
7. Christian Bunse, Hans-Gerhard Groß, and Christian Peper. Embedded system construction - evaluation of model-driven and component-based development approaches. In *MoDELS Workshops*, pages 66–77, 2008. Best Workshop Paper Award.
8. Christian Bunse, Nicole Levy, and Felix Freiling˙A Taxonomy on Component-based Software Engineering Methods. In Ralf Reussner, Judith Stafford, and Clemens Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume LNCS 3938. Springer, 2003.
9. Laura Campbell, Betty Cheng, William McUmber, and R.E.K. Stirewalt. Automatically detecting and visualising errors in UML diagrams. *Requirements Engineering*, (7):264–287, 2002.
10. John Cheesman and John Daniels. *UML Components - A Simple Process for Specifying Component-based Software*. Addison-Wesley Longman, Amsterdam, 2000.
11. Yunja Choi. Checking interaction consistency in MARMOT component refinements. In *Proceedings of SOFSEM 2007, LNCS 4362*, January 2007.
12. Yunja Choi. Verification of an abstract component using communication patterns. In *2009 ICSE Workshop on Model-based Methodologies for Pervasive and Embedded Software*, May 2009.
13. Yunja Choi and Christian Bunse. Towards component-based design and verification of a μ-controller. In *11th International Symposium on Component-Based Software Engineering*, pages 196–211, 2008.
14. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
15. Edmund M. Clarke, Jeannette Wing, and et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.
16. Desmond D'Souza and Alan C. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.
17. Steve EasterBrook, Robyn Lutz, and Richart Covington et al. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, January 1998.
18. Gregor Engels, Jochen M. Kuester, and Luuk Groenwegen. Consistent interaction of software components. *Journal of Integrated Design and Process Science*, 6(4):2–22, December 2003.
19. Luis Gomes et. al. Towards usage of formal methods within embedded systems co-design. In *10th IEEE International Conference on Emerging Technologies and Factory Automation*, September 2005.
20. M. Fung, Brian Henderson-Sellers, and L.-M. Yap. A comparative evaluation of OO methodologies from a business rules and quality perspective. *Australian Computer Journal*, 29(3):95–101, 1997.
21. Gerald C. Gannod, Robyn R. Lutz, and Marian Cantu. Embedded software for a space interferometry system: Automated analysis of a software product line architecture. In *IEEE International Conference on Performance, Computing, and Communications*, April 2001.
22. D. Gay, P. Levis, and R. Behren et al. The nesC language: A holistic approach to networked embedded systems. In *Conference on Programming Language Design and Implementation*, pages 1–11, June 2003.
23. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the Computer Aided Verification(CAV 1997)*, pages 72–83, 1997.
24. Object Management Group. UML2.0 superstructure specifications.
25. Hermann Haertig, Steffen Zschaler, and Martin Pohlack et al. Enforceable component-based realtime contracts: Supporting realtime properties from software development to execution. *ACM Transactions on Software Engineering and Methodology*, 2007.
26. Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series., 1991.
27. Gerard J. Holzmann. *The* SPIN *Model Checker : Primer and Reference Manual*. Addison-Wesley Publishing Company, 2003.
28. Jozef Hooman, Hillel Kugler, and Iulian Ober et al. Supporting UML-based development of embedded systems by formal techniques. *Software Systems Modeling*, 2008.
29. Pao-Ann Hsiung. Formal synthesis and code generation of embedded real-time software. In *9th International Symposium on Hardware/Software Codesign*, April 2001.
30. Daniel Jackson and Jeannette Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.
31. J.Hill, R. Szewczyk, and A. Woo et al. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, November 2000.

32. Steven D. Johnson. Formal methods in embedded design. *IEEE Computer*, November 2003.
33. C. Kern and M. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of E. Systems*, April 1999.
34. M.U. Khan and K. Geihs et al. Model-driven development of real-time systems with UML 2.0 and C. In *Proceedings of the 3rd International Workshop on Model-based Methodologies for Pervasive and Embedded Software at the 13th IEEE Int. Conf. on Engineering*, 2006.
35. Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, pages 637–664, 1999.
36. Formal Systems Europe Ltd. Failures-divergence-refinement: FDR2 user manual, 1997.
37. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
38. Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard Holzmann. Implementing statecharts in PROMELA/SPIN. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, October 1998.
39. Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
40. Bill Mitchell. Characterizing comminication channel deadlocks in sequence diagrams. *IEEE Transactons on Software Engineering*, 34(3):305–320, May/June 2008.
41. M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.
42. Gustaf Naeser and Kristina Lundqvist. Component-based approach to run-time kernel specification and verification. In *17th Euromicro Conference on Real-Time Systems*, 2005.
43. Ileana Ober. Action specification in OMEGA, 2004. Technical Report, Verimag, http://www-omega.imag.fr/.
44. Oscar R. Ribeiro, Joao M. Fernandes, and Luis F. Pinto. Model checking embedded systems with PROMELA. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, 2005.
45. John Rushby and David W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report CSL-95-10, SRI International, August '96.
46. Clemens Szyperski. *Component Software: Beyond Object-oriented Programming*. Addison-Wesley Publishing Company, 2002.
47. Mircea Trofin and John Murphy. Static verification of component composition in contextual composition frameworks. *Software Tools and Technology Transfer*, 2008.
48. W.M.P. van der Aalst, K.M. van Hee, and R.A. van der Toorn. Component-based software architectures: A framework based on inheritance of behavior. *Science of Computer Programming*, 42(2-3), 2002.
49. Fei Xie and James C. Browne. Verified systems by composition from verified components. In *Proceedings of Joint Conference ESEC/FSE*, 2003.
50. Wooseung Yang, Moo-Kyeong, and Chong-Min Kyung. Current status and challenges of soc verification for embedded systems market. In *IEEE International Conference on System-On-Chip*, 2003.
51. Tewfik Ziadi, Loïc Helouët, and Jean-Marc Jezequel. Revisiting statechart synthesis with an algebraic approach. In *26th International Conference on Software Engineering*, 2004.