Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Matěj Outlý

Mode Change in Real-time Component Systems

Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Bureš, PhD.
Study program: Computer Science, Software Architectures

2011

I would like to thank Tomáš Pop for thorough supervision and numerous suggestions and Tomáš Bureš for inspiration and useful discussions.

Název práce: Změna módů v real-timových komponentových systémech
Autor: Matěj Outlý
Katedra (ústav): Katedra spolehlivých a distribuovaných systémů
Vedoucí práce: RNDr. Tomáš Bureš, PhD.

Abstrakt: Cílem práce je zmapovat možnosti dynamické rekonfigurace v real-timových komponentových systémech, především pak formálně popsat podporu operačních módů. Práce obsahuje návrh mechanismu, který zajišťuje rekonfiguraci na základě řídících proměnných a jejich vzájemných vztahů. Mechanismus je navržen tak, aby bylo možné snadno namodelovat operační módy a pravidla pro rekonfiguraci při zachování znovupoužitelnosti komponentového návrhu. Práce dále ukazuje způsob realizace tohoto mechanismu tak, aby byl použitelný ve světě vestavěných real-time systémů.
Klíčová slova: rekonfigurace, operační módy, real-time systémy, komponenty

Title: Mode Change in Real-time Component Systems
Author: Matěj Outlý
Department: Department of Distributed and Dependable Systems
Supervisor: RNDr. Tomáš Bureš, PhD.

Abstract: The goal of the thesis is to examine possibilities of dynamic reconfiguration in real-time component systems, especially to formally describe support of operating modes. The thesis introduces a reconfiguration mechanism based on properties and relations between them. The mechanism is designed to facilitate a straight forward modeling of operating modes and reconfiguration rules and preserves reusability of assembled components. The thesis also presents a realization of the mechanism suitable for the domain of embedded real-time systems.
Keywords: reconfiguration, operating modes, real-time systems, components

Contents

List of Figures

Chapter 1

Introduction

1.1 Real-time Applications

Embedded devices are used in different branches of business or science, from spacecraft engineering, avionic and automotive industry to home automatization. An important sort of the devices is control systems. A control system is a single-purpose real-time system deployed at limited hardware. Its behavior must be predictable and the execution must meet all applied time-related constraints, thus the device must respond to an event in a limited time. Another constraint can be e.g. limited memory, thus the system must have a predictable memory footprint.

Recently complexity in the domain of real-time systems has been growing [4]. This is reflected in the application's development. The more complex the application the more difficult the development and the more skilled programmer has to develop it. Therefore, new techniques to enable easier, faster and less expensive development are researched. The problem can be solved by introducing a concept providing some boundaries and making possible to re-use some parts of the application.

A promising solution for the problem of growing complexity lies in component-based development. This approach uses a common library to store different application parts – components. Components are created with respect to re-usability and compatibility with each other. When a particular application is developed, only a new logic is created and encapsulated in new components. The rest of the application is assembled from the existing components loaded from the common library. This development technique is appropriate for real-time applications as well. However, real-time constraints and constraints related to limited hardware must be kept in mind. The methodology must be adjusted accordingly.

1.2 Application Variability

With the application complexity growing there comes an effort to create it more variable. There are different kinds of variability. It is possible to study variability at design time. An application can be designed with respect to easier later modifications. Methods enhancing application's flexibility, e.g. design patterns, are introduced. Variability can be observed at compile time as well. Application's behavior can be modified through a compiler configuration, e.g. by preprocessor macros.

The thesis refers to variability of real-time component systems at runtime, mostly an ability to dynamically reconfigure the application according to the situation it lies in. A control system interacts with the real world that can be heterogeneous. The device has to be flexible enough to properly react on accidental changes in the environment. Hence the runtime reconfiguration is a natural request to control systems.

General forms of runtime reconfiguration are not suitable for real-time embedded devices because they are limited in terms of computational time and memory. It must be assured that the resources are not overdrawn in any reachable configuration the application lies in. One possible solution lies in restricted forms of reconfiguration. Only a limited set of configurations is described in advance. The application is built for the selected configurations and it is assured that the resources are sufficient for every single configuration (so called an *operating mode*).

1.3 Goal and Structure

The goal of the thesis is to design and formally describe a support for operating modes in a real-time component-based application with respect to preserving basic characteristics – re-usability of components and operating modes and schedulability of the application and each of its configurations. The thesis presents methodology to construct data structures enabling predictable and effective mode change handling at runtime. Last but not least a prototype of the proposed extension is implemented as a proof of the concept.

The thesis is structured as follows. Chapter 2 and Chapter 3 provide theoretical background about real-time systems and component-based development which is necessary for understanding the topic. Chapter 4 analyses possible methods to complete the goals and creates a summary of new ideas introduced in the thesis. Chapters 5, 6 and 7 complement the analysis with formal definitions and detailed description. The concept implementation is described in Chapter 8. Chapter 9 gives an evaluation of the concept using a case study application. Work related to the topic is summarized in Chapter 10 and the thesis is concluded by Chapter 11.

Chapter 2

Real-time Application

Based on [3], [2] and [5], this chapter briefly introduces a concept of real-time systems. Basic terms are described in the first part. The second part of the chapter summarizes basic facts about reconfiguration and mode change protocols. The third part briefly outlines a question of dynamic memory allocation.

Real-time system is a system where not only a correct result but also the time when it is delivered is important. The definition does not mean that the result should be returned quickly but more likely on time. This time constraint is often denoted as a deadline.

The thesis aims at *hard* real-time systems, i.e. systems where returning a result after deadline causes application's breakdown and catastrophic consequences on the controlled environment.

2.1 Tasks and Scheduling

A real-time system can integrate several different activities – *tasks*. The activities together represent behavior of the system. Each task is characterized by the following parameters.

- *Arrival time* (a) is the time when a task becomes ready for execution.
- *Computation time* (C) is the duration how long a task is executed.
- *Absolute deadline* (d) is the time before which a task should be completed. Absolute deadline can be replaced by *relative deadline* (D) which is the difference between absolute deadline and arrival time.
- *Start time* (s) is the time from which a task is being executed.
- *Finish time* (f) is the time when a task is completed.
- *Response time* (R) is the difference between finish time and arrival time.

More parameters are defined in [3]. However, since they are not relevant to the thesis, their description is omitted.

A task can be either periodic or aperiodic. A *periodic* task consists of an infinite sequence of identical activities, called *instances*, activated regularly after a time *period* (T). An *aperiodic* task consists of an infinite sequence of instances as well, but they are not activated on a regular basis. Just systems with periodic tasks are considered in the thesis. The described architecture, denoted as a *Time Triggered Architecture* (TTA), is considered to be suitable for hard real-time systems development.

The main objective of real-time systems is the way how to schedule task's instances in order to meet all task's deadlines ($f \leq d$), to find a *feasible schedule.* A system implements a *scheduling algorithm* for this purpose. A scheduling algorithm often works with priorities. A task's *priority* (P) is estimated on the basis of the period or the deadline. The priority affects which waiting task is executed as first. During system's development it is possible to use a *schedulability analysis* which detects if the chosen scheduling algorithm is able to properly schedule the set of tasks.

Scheduling algorithms can be classified by different criteria.

- *Preemptive* or *non-preemptive* algorithm – An algorithm is preemptive if it interrupts an instance's execution, otherwise it is non-preemptive.

- *Static* or *dynamic* parameters – An algorithm handles dynamic parameters if they can vary over time. Parameters are static if they are fixed all the time the system is running.

- *On-line* or *off-line* algorithm – An algorithm is on-line if it is applied at runtime every time the system needs to switch tasks. It is off-line if a schedule is precomputed in advance and stored in a table.

Frequently used scheduling algorithms are *Rate Monotonic* (RM) and its modifications and *Earliest Deadline First* (EDF). RM is a static on-line algorithm applicable for periodic tasks. EDF is a dynamic on-line algorithm for periodic tasks scheduling.

There is a set of parameters which completely characterizes a periodical task in the system. It is a computation time, a relative deadline and a period. The three parameters are constant for all instances. A relative deadline, a period, a computation time estimated by a *worst case execution time analysis* (WCET) and a priority computed by a scheduling algorithm are called *real-time attributes.* The real-time attributes are used for task's configuration in the system.

A software part of a real-time system usually consists of a *real-time operating system* (RTOS) and a *real-time application.* The real-time operating system handles low-level issues like scheduling and memory management and offers specific real-time services like support for periodic tasks. There exists several real-time operating systems, e.g. FreeRTOS [19] or some real-time modification of GNU/Linux [20]. The real-time application, on the other hand, implements the system's specific issues and creates its business logic.

2.2 Runtime Reconfiguration

General forms of runtime reconfiguration are not suitable as described in Chapter 1. Operating modes has been outlined as a possible solution. An operating mode is a configuration described in advance which corresponds to specific behavior of the system. In principle, it defines which tasks are running and which values of real-time attributes are applied to each running task. This section provides details about operating modes and transitions between them. A piece of knowledge contained in the entire section origin from [2] where exhaustive description of the topic can be found.

In a multi-moded application the previous definition of a task is not sufficient and must be extended. Real-time attributes may vary between different operating modes. Thus a periodic task is described by tuples of a computation time, a period, a deadline and a priority, one for each operating mode in the system. A task may or may not be active in a particular operating mode.

A *mode change request* (MCR) is an event which causes a transition from an old operating mode to a new one. It can occur in the steady state when a particular mode is active, but not when a transition is in progress. The application should ensure that this request is fulfilled or introduce a mechanism which handles it.

Tasks involved in a mode change can be divided to *old-mode* and *new-mode* tasks. Old-mode task can be:

- *Aborted* at MCR if the abortion does not break a data consistency.
- *Completed* during the mode change.

New-mode task can be:

- *Wholly new* if it is not active in the old mode.
- With *changed real-time attributes* if it is active in the old mode, but with different values of real-time attributes.
- *Unchanged* at all if it is active in both old and new modes with the same values of real-time attributes.

The application can be overloaded during a mode change. Some old-mode tasks have to be completed and new-mode tasks have to be activated in the same time interval. In order to prevent overdrawing of available resources, the first activation of new-mode tasks have to be delayed. The delay relative to the MCR is called an *offset*. An offset of a task is defined for each possible mode change.

Since offset may delay the activation of a new-mode task, there is a requirement to assure its promptness. Each task defines a *mode change deadline* which bounds task's response time during mode change for this purpose. A mode change deadline is the maximum time allowed for the first activation of the task in the new mode to be completed, with respect to MCR.

The main objective is to find appropriate offsets for each task in the system with respect to mode change deadlines. An algorithm which finds it is called a *mode* *change protocol*. Existing protocols can be classified according to different characteristics.

- Protocols *with periodicity* or *without periodicity* – In a protocol with periodicity unchanged tasks are executed independently from the mode change. In a protocol without periodicity the activation of unchanged tasks may be delayed.

- *Synchronous* or *asynchronous* protocols – In a synchronous protocol new-mode tasks are released after old-mode tasks are completed. In an asynchronous protocol a combination of both new-mode and old-mode tasks is allowed during the mode change.

Maximum-period offset protocol is one of the most trivial mode change protocols. This synchronous protocol delays all new-mode tasks for the time equal to the maximum period of both old-mode and new-mode tasks. Unchanged tasks are not affected, thus the protocol is with periodicity. It is very simple to implement but suffers with poor promptness. Better promptness can be assured by incorporating one of the asynchronous protocols. However, implementation of an asynchronous protocol is not so simple and a special analysis to validate that the resources are not overdrawn during transition goes with it.

2.3 Dynamic Storage Allocation

Dynamic memory allocation suffers with time unpredictability during both allocation and deallocation operations and with a problem of unknown memory size allocated at runtime. Another problem is memory fragmentation which can cause unreliable services, especially when the application runs for a long period of time.

The problems can be solved by introducing a special allocator bounded to fulfill requirements to real-time systems. Algorithm used by such an allocator is called a *dynamic storage allocation* (DSA) algorithm. Bounded and fast response time and bounded and low memory fragmentation are the common requirements. Example of such an algorithm can be *Half-fit* or *TLSF* algorithm. More information about the topic can be found in [5]. However, in order to use DSA allocator it has to be implemented in the used RTOS. It is a feature which cannot be relied on.

Chapter 3

Hierarchical Component System

This chapter introduces a component-based development, not in a comprehensive form useful for praxis but only in a form sufficient for the thesis. Exhaustive introduction to the topic can be found in [6]. A hierarchical component system is considered.

A *hierarchical component system* is a development framework which allows to create independent components or other application parts and store them in a *common library*. The components are composed together creating a business logic of developed application. They can be nested to each other at different levels creating a hierarchy. When the design is finished, the application is deployed and prepared to be launched.

An example of a hierarchical component system is SOFA 2 Component system [12] or Fractal Component Model [13]. There exist many different component systems. Each component system can significantly differ from one another. A general idea of a component system is described and assumed in the thesis with the following characteristics.

- Existence of a meta-model defining hierarchical system of components.
- Using an architecture description language for application model description.
- A component life-cycle composed of design, deploy and execution phase.
- Existence of active components with real-time attribute values defined.
- Existence of schedulability analysis.

It is not assured that the idea is coherent with all existing component systems but for the most of them it is applicable in the presented or in a rather modified form.

3.1 Meta-model

In modern component systems, an application is specified as an instance of a *meta-model*. Meta-model is a model which defines particular building blocks and supporting structures of a developed application. It describes how a component looks like, how an interface is created for a component, how components can be connected or composition rules which must be kept. In praxis, a meta-model is designed using a modeling language, e.g. UML [17] or EMF [18]. The ideas presented in the thesis assume the meta-model described as follows.

The main building block of a developed application is a *component.* At the first sight a component is a black box which exposes a collection of *interfaces.* An interface is an entity that creates an apposing point allowing the component to communicate with its surrounding.

A component is either primitive or composite. A *composite* component contains other components as *subcomponents* and defines *connections* between their interfaces. The component's business logic is created by the composition and by the *component source code*, if it is defined. On the other hand a *primitive* component does not contain any other elements and its business logic is encapsulated in the *component source code.* Figure 3.1 shows an example of a composite component with two subcomponents and connections between their interfaces.

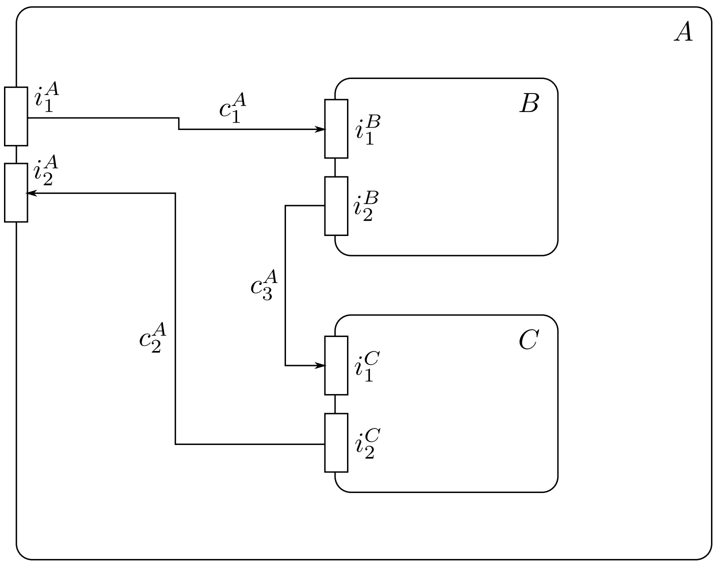


Figure 3.1: Example of a single composite component with two subcomponents

Interfaces are of two kinds. *Require* interfaces require a service from another component. *Provide* interfaces provide a service to another component. An interface can be connected to *compatible* interface of other component so data can be transfered from one interface to another. The first compatibility aspect is an interface *data type.* The data type of the connected interfaces has to be the same. The second aspect is the interface kind. Require interfaces are compatible with provide interfaces of sibling components. Provide interfaces of parent components are compatible with provide interfaces of subcomponents. Require interfaces of subcomponents are compatible with require interfaces of the parent component.

A *configuration* of a component is realized using configuration parameters. A set of parameters is defined during the component design. However, values of the parameters can be set later before the component is launched. The parameter values can be used by the source code or by another entity that understands the parameter's semantics to adjust the component's behavior.

A set of methods, called a *component API*, is available to the component source code. It enables the source code to work with the component's interfaces and configuration parameters.

An *application* is represented by one *top-level component* which encloses all other components and their connections. A component is *contained* in the application if it is a direct or indirect subcomponent of the top-level component.

3.2 Structural Requirements

Well defined application has to hold the following *structural requirements.*

(1) An arbitrary require interface of an arbitrary component has to be connected to exactly one compatible provide interface of a sibling component or to exactly one compatible require interface of a parent component.

(2) An arbitrary provide interface of a composite component has to be connected to exactly one provide interface of a subcomponent.

There is a corollary directly implied from the structural requirements. The requirements ensure that a thread of execution which is currently in an arbitrary component can call an arbitrary require interface of the component and this interface is connected to a provide interface of another component where the execution can continue.

3.3 Architecture Description Language

An *architecture description language* (ADL) is used for the component-based application development, i.e. creating a meta-model instance. It is a language which enables to design, configure and validate an application and acts like a middleman between the user and the component system.

3.4 Application Life-cycle

The development of an application can be divided into three basic phases – design phase, deploy phase and execution phase.

In a *design phase*, already created components are loaded from the common library and new components are created. An architecture description language is used for creating an application structure and implementation language is used for component's source code writing.

When the application design is complete, a *deploy phase* begins. In this phase, a designed application is analyzed, component instances are created and a control part of application's source code is generated. Some component systems incorporate flat component model for deployed components instead of the hierarchical one. Advantage of this approach is an easier application structure at runtime in exchange for a more complex deployment process.

In the last phase the deployed application is launched and debugged. The time when the application code is executed is called *runtime.* The application can be stand-alone or it can run with an assistance of the component system or a virtual machine.

3.5 Real-time Extension

A specification of a hierarchical component system has to be extended in order to be suitable for the real-time application's development. This section presents a few concepts used in such an extension.

The application have to keep the paradigm of TTA that has been chosen in Chapter 2 as a suitable model for real-time application's development. Every instance of a periodic task is one time trigger. All instances of one particular task accomplish one particular activity, thus the task require one entry point to the application where the activity is implemented. For the sake of simplicity, a component defines at most one entry point. A component which defines an entry point is called *active.*

Values of *real-time attributes* have to be defined for each active component in order to configure its real-time behavior at runtime. The attribute values are not a part of the application's structure, but more likely a part of the configuration. Thus some of the configuration parameters can be reserved for the real-time attributes. As it is used by a schedulability analysis it has to be defined before the application is deployed.

A deploy phase can incorporate a *schedulability analysis.* It verifies if it is possible to schedule the designed application on the target resources. First, the analysis computes worst case execution times of active component tasks. Then it uses this information together with values of real-time attributes, information about target resources and intended scheduling algorithm to compute a result. Based on the result the application is either deployed or an error is reported.

Since the application is distributed with RTOS, there is a natural request for a configurable choice which operating system is used. Introducing a common application interface that covers differences between operating systems can be a part of the implementation. The interface can offer specific low-level services related to the operating system as well. It is called a *system API.*

Chapter 4

Analysis

This chapter analyzes possible methods to design and implement concept of mode change in real-time component systems as introduced in Chapter 1. The selected method is discussed in Chapters 5, 6 and 7 in a more formal way.

4.1 Modes in Component System

Reconfiguration of a monolithic real-time application is described in Chapter 2. The important conclusion of the chapter is that the application defines a restricted set of well known configurations – operating modes. This section discusses exact meaning of operating modes in the context of the component-based development.

One of the most important advantages of the component-based development is strict encapsulation. When a component is being developed, the scope of the whole application does not have to be considered. It can be aimed at the particular component only, thought at one level of abstraction and thus the component design is clean and simple. It is desirable to preserve the component encapsulation even if the operating modes are included.

For this reason, each component can define its own set of *component modes*. For different components, sets of component modes are independent and a relation between them is not established until components are composed into a specific application. Therefore, a set of component modes can be encapsulated together with a component. A transition between two modes in a set of component modes is invoked on the basis of a *mode change event*.

Each component mode defines a configuration and represents behavior of the component only, without any dependence on the rest of the application. A set of component modes is meaningful for two kinds of components – composite components and active components.

Composite Component Internal structure of a composite component is created by subcomponents and connections between them. A mode change of a composite component is in fact a modification of this structure. It can be viewed as a process when new subcomponents and connections are enabled and the old ones are disabled.

A subcomponent or a connection can be classified according to its behavior during mode change.

- *Old-mode subcomponent/connection* – It is enabled in the old mode but disabled in the new one.
- *New-mode subcomponent/connection* – It is disabled in the old mode but enabled in the new one.
- *Unchanged enabled subcomponent/connection* – It is enabled in both old and new modes.
- *Unchanged disabled subcomponent/connection* – It is disabled in both old and new modes.

Active Component A mode change of an active component can adjust values of real-time attributes of its task. The component can be classified according to it.

- *Changed real-time attributes* – Real-time attributes of the component's task are adjusted in the new mode.
- *Unchanged real-time attributes* – Real-time attributes of the component's task are the same in both old and new mode.

4.2 Mode Change Mechanism

A mode change process can be realized by a *mode change mechanism*. It is a set of rules where and under what circumstances a mode change event arises. Part of the mechanism is a specification of data structures and algorithms implementing the mechanism rules. There are several requirements to the mechanism. An overview and origin of each requirement follows.

Component Independence and Re-usability (R1) One of the main characteristics of a component system must be preserved when a mode change mechanism is applied. Components need to be independent and re-usable in order to be stored in the common library and re-used for another application's development. This requirement is crucial.

Not Polluting the Existing Concept (R2) This requirement refers to a component's source code and interface. If a component's source code were polluted with a mode management code, it would cover useful business logic and the source code would be significantly more complicated. A similar situation can be observed for for component's interfaces which could not be polluted with auxiliary mode-change specific services. This requirement is considered optional.

Predictable Overhead at Runtime (R3) The main requirement to real-time applications has to be preserved for the implementation of a mode change mechanism at runtime. The mechanism's overhead needs to be predictable from both time and memory points of view. This requirement is crucial.

There are several basic propositions for a mode change mechanism ensued directly from the requirements. The propositions are shown in Figure 4.1 and listed and described as follows.

Local Definition of a Mechanism (P1) This proposition refers to the place where a decision about the active mode is made. A component has to decide about itself locally in order to meet the first requirement. If some other entity made this decision for the component, it would be dependent on the entity and it could not be easily stored in the common library and re-used later.

Communication on Components (P2) There are aspects affecting which component mode is active. The aspects can be divided by the source of origin. They can come from a component itself (the mode depends on a component's actual state). But the aspects can also come from a subcomponent (the mode depends on a state of component's assemblies). The last source of origin is the component's outer surrounding.

Assuming the previous proposition that a component's mode change mechanism is defined locally, there has to be a way how to gain this data from both inner and outer surroundings. Therefore, a communication between a parent component and subcomponents has to be allowed in an assembled application.

Mode Condition (P3) Assuming the previous propositions, a component's mode change mechanism is about to decide which mode is active based on the data gained from the inner and outer surroundings and the component itself. The data are evaluated by a mode condition. The mode condition then returns a component mode that needs to be set as active.

An example of straight forward but inconvenient mode change mechanism is introduced in the next section. Learning from its drawbacks, a proposition of a mechanism which fulfills all requirements and follows all propositions is described later.

4.3 Manual Mechanism

The simplest way to manage triggering mode change events is to delegate responsibility to the user, to let him manually trigger a mode change event from a source code of a component, e.g. through a component API.

This approach has several drawbacks. There is a natural requirement to be able to emit a mode change of a component from its source code. But what about com-

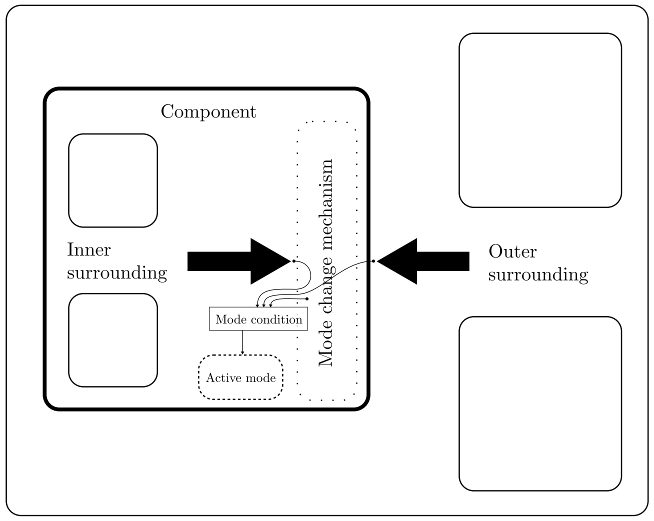


Figure 4.1: Mode change mechanism basics

ponents which do not contain any source code? Such a component can be managed either from a subcomponent or from a parent component. The first case means that the subcomponent cannot be independent from its parent component. The second case means that the parent component has to know details about the contained subcomponent, especially which modes it defines and what the modes mean. Both situations break the mechanism requirement (R1).

The mode change specific communication described in the proposition (P2) is hard to be satisfied. The communication is supposed to be implemented manually by introducing new auxiliary component interfaces and connecting components through them. It breaks the requirement (R2) and brings a lot of worthless work.

The similar drawback is caused by the mode condition described in the proposition (P3). This mechanism places the condition to a component's source code. It also breaks the requirement (R2).

Last but not least, responsibility for the runtime overhead is delegated to the user as well. The user has to be aware of this issue. If he is not, the requirement (R3) can be easily broken.

4.4 Property Based Mechanism

To fix the drawbacks of the manual approach, a mechanism can be automatized and integrated to the component system. It means that the definition of a component is extended by a possibility to describe the mode change mechanism rules, not defined in the component's source code. The component system implements algorithms which realize the mechanism rules. The only responsibility of the component is to exhibit relevant data into the mechanism. This section proposes such an automatic mechanism.

4.4.1 Basic Design

A set of common *properties* for each component can be defined. The properties are memory fragments reserved for a component to expose a component's internal state. Properties have two main responsibilities.

- To distribute the mode change relevant information across the application.
- To be input for the component's mode condition.

Properties from different components are connected and values are transferred between them. The connections can be made across the whole application. The mode change relevant information is carried by properties as property values and distributed across the application through the connections. Thus properties act like a medium for an application-wide mode change handling in order to fulfill the first responsibility.

Assume that a component uses a relevant data to decide which mode is active. The data is transferred to the component through properties and their connections as described. Then it is stored as a property value in the component. The component's mode change mechanism uses the values as input for a mode change condition. Thus the second responsibility is ensured.

Moreover, the component mode itself can be one of the component properties. An advantage of this design is the following. The component mode itself can be used as input for the condition or even for computing a value of other property. There can be a situation when the new mode depends on the current one. This design makes this situation possible to describe.

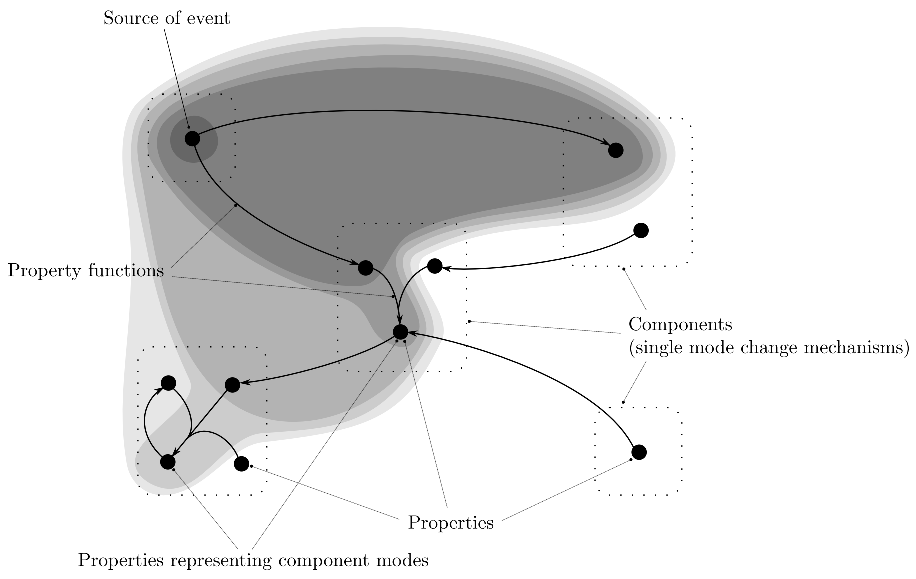

Source of event
Property functions
Components
(single mode change mechanisms)
Properties
Properties representing component modes

Figure 4.2: Event spreading through a property network

The connections can be made using *property functions*. A property function is an arbitrary function defined from an input set of properties to a single output property. It can be a function of one or more variables. If a combination of input values is not in a domain of a function then the function takes no effect and a value of output property is not changed.

Properties with functions create a *network*. When a value of some property is assigned from the outside, the network reacts as property functions describe. The event spreads through the network until it is stabilized. The network stabilization is not always assured, it depends on the designed network structure. When the network is stabilized a possible component reconfigurations are made according to values on the properties. The network reaction and the issue of its stabilization is described in detail in Section 4.4.3. An example of an event spreading through a network is shown in Figure 4.2.

4.4.2 Meta-model Extension

Implementing modes and a mode change mechanism brings some new elements which must be included to the meta-model. In any case a component must define a set of modes. Besides this, a mode change mechanism specific elements must be added. The new elements are mode properties and property functions.

A property function can be observed at two different levels. At an abstract level it is a template of function's behavior. It is the description how input values are rewritten to an output one. But there's nothing said about actual properties which the function is applied on. A second level is an instantiation to actual properties in the network. At this level, it is possible to describe a relation between properties and specify which function is applied. It is possible that one property function is applied in many different places in the network.

This dividing is optional but very useful. Therefore the meta-model contains two parts – a definition of property function templates and a possibility to instantiate them for different properties.

The application structure is specified using the architecture description language. But there is a need to specify function's behavior. It is not advisable to pollute ADL with it. Moreover, it is a task at a lower level of abstraction. A new language, fully independent on ADL, can be introduced. Let us call this language a *property function language* (PFL).

4.4.3 Network Reaction

The property based mechanism is suitable from the user point of view. It enables the user to effectively model independent component modes without polluting source code or component interface. The rest of the analysis revises the mechanism from the implementation point of view. This section analyses the network reaction.

Assume a property network. Each property has assigned a value from its domain, i.e. properties are in a *property state*. The network reaction begins with assigning a new value to a property, called an *initial property*, creating a new property state.

Even if the new value is not different to the old one, the network reacts because it can cause modifications of property values dependent on the initial one. Example of such a situation can be found in Figure 4.3. Once the new value is assigned, an event is spread transitively following all property functions which takes the initial property on input. Such functions have an output property. The function is invoked with the property state on input to compute a new value of the output property. Assigning the computed values to the output properties creates a new property state and the whole process can iterate until the network is stabilized. When the network is stabilized the property state it lies in is called *stable*.

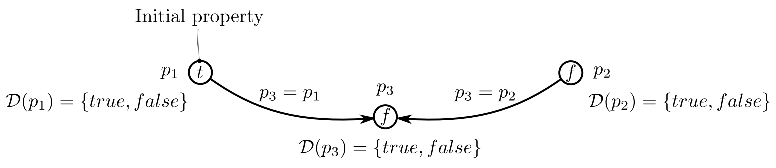

Initial property
p_1
t
$\mathcal{D}(p_1) = \{true, false\}$
$p_3 = p_1$
p_3
f
$\mathcal{D}(p_3) = \{true, false\}$
$p_3 = p_2$
f
p_2
$\mathcal{D}(p_2) = \{true, false\}$

Figure 4.3: Network where assigning a current value to the property p_1 causes modification of the dependent property p_3

There are no restrictions in the network, e.g. it can contain cycles. Cycles can cause oscillation in the network reaction and thus are the major problem which must be handled. Oscillation arises when a property value is alternated in a cycle and never stabilized. An example of such oscillation is shown in Figure 4.4.

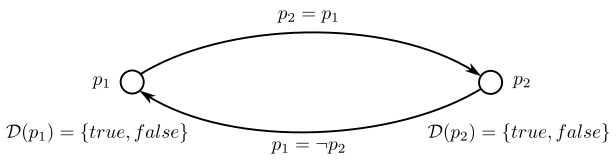

$p_2 = p_1$
p_1
p_2
$\mathcal{D}(p_1) = \{true, false\}$
$p_1 = \neg p_2$
$\mathcal{D}(p_2) = \{true, false\}$

Figure 4.4: Network whose reaction always oscillate

Oscillation depends on the network structure and on a property state. There are cases when the network oscillates during reaction with one particular property state on input and with a different one it does not, as shown in Figure 4.5.

The mode change mechanism cannot work properly with a network which oscillates in an accessible case. When the user designs an astable network, development tools should alarm him and an application cannot be compiled.

There may be possibility to introduce restriction rules for a network design which makes impossible to design an astable network. The rules can restrict either a structure or a property function semantics. One of the reasonable restrictions of the structure is to forbid cycles. The only practicable restriction of the property function semantics is to allow a value propagation ($p_2 = p_1$) only. But both the rules are restrictive for a praxis and, as shown in the rest of the thesis, are not necessary. They degrade the mechanism too much that it would not be operational with them.

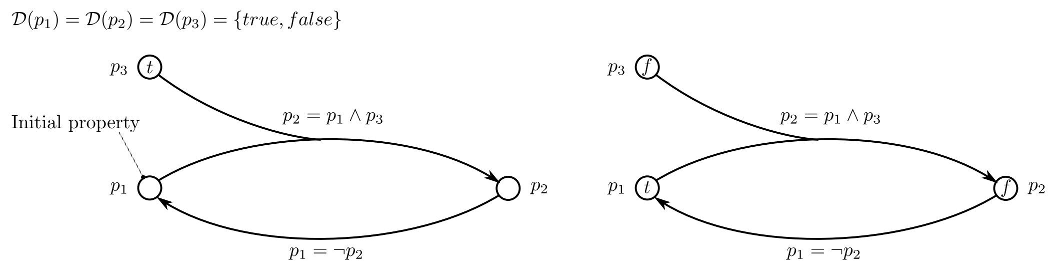


Figure 4.5: Network where oscillation depends on a property value

Let us introduce an algorithm, called a *react* algorithm, which represents the network reaction. The algorithm has to return the same result as the network reaction (described before) would. Besides the correct representation, it must finish in finite time and answer if the reaction oscillates or not. It is quite unpleasant situation because it makes the algorithm impossible to work with time complexity polynomial to the number of properties.

The first version of the react algorithm can be following. A network is fully represented in memory and a reaction is simulated on it using standard BFS algorithm. A property state encapsulated with a current token stands between two algorithm steps. A token is a set of properties which have been modified in the previous step and which are considered to be used as input properties in the next step. The property states with tokens have to be stored, e.g. in a hash table, in order to avoid their repeated visiting. If a property state with a token is visited for the second time, it means that the network oscillates. If a simulation does not change property values anymore, it means that a stable state is found. For this reason, the algorithm has time and space complexity, in the worst case, exponential to the number of properties.

4.4.4 Resemblance with SAT

Let us try to construct the react algorithm by transformation to an instance of the SAT problem. It seems that the problems are analogous and the transformation is possible. This section presents transformation possibilities and shows that the transformation is not as straight forward as it seems.

The first SAT version of the react algorithm incorporates a transformation that creates a variable for each property in the network and models relations between variables resembling to property functions. The SAT instance have to construct a set of affected properties which contains all properties modified by the network reaction. Each property function with input properties p_{i_1} and p_{i_2} and an output property p_o is modeled by the following facts

$$p_{i_1} \in Af \implies (p_o = f(p_{i_1}, p_{i_2}) \wedge p_o \in Af)$$
$$p_{i_2} \in Af \implies (p_o = f(p_{i_1}, p_{i_2}) \wedge p_o \in Af)$$

With a fixed value of the initial property and the initial property included in Af, a solver finds evaluation of all affected properties which holds all property functions.

The first SAT version has a major drawback. If the network contains cycles it can return incorrect results as defined in the previous section. It is caused by a temporal character of property functions. It means that they do not necessarily hold in a stabilized network, but they must hold in some step that precedes the network stabilization. However, the SAT instance enables only a situation when all property functions hold on the affected properties at the same time.

The second SAT version of the react algorithm tries to correct the first version's problem. Each property is also represented by a single variable and property functions are represented by relations between the variables. Besides this, a solver gets additional information about order in which the reaction spreads through the network. A solver decides which property functions hold and which do not on the basis of this information. The order can defined as in 5.16. Assume that a number $order(p)$ defines the order of a property p. A solver is unable to compute the set of affected properties by itself, thus it has to be provided on input, e.g. modeled by the relation $p \in Af \Leftrightarrow order(p) = 0$. Then assume that a property p_o is output of functions $f_1(p_{i_1})$ and $f_2(p_{i_2})$. The facts modeling the functions of property p_o are following.

$$\begin{array}{l} order(p_{i_1}) > 0 \ \wedge \ order(p_{i_1}) \geq order(p_{i_2}) \Longrightarrow p_o = f_1(p_{i_1}) \\ order(p_{i_2}) > 0 \ \wedge \ order(p_{i_2}) \geq order(p_{i_1}) \Longrightarrow p_o = f_2(p_{i_2}) \end{array}$$

A solver finds evaluation of all affected properties as follows. Each property is computed by the function which is the last in the input order. Functions computing the property before the last function do not necessarily hold.

A drawback of the second SAT version is that a solver can find more that one evaluations satisfying the definition of stable state. This effect is shown in Figure 4.6. But only one of them is the stable state resulting from the computed network reaction. The second phase of the algorithm have to be a decision about the right result. It can be done by simulating the network reaction and waiting for a property state that is among the solver's results. This drawback is caused by the fact that the SAT instance does not model the whole process of the network reaction but only its result.

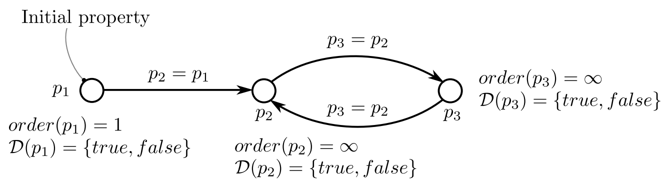


Figure 4.6: Network where the second SAT transformation founds two results

The last presented SAT version of the react algorithm incorporates a history of the network reaction. For this feature, each property has to be represented by as many variables as is the number of unique steps in the reaction. Besides property values, also tokens (described in the previous section) have to be represented in each reaction's step. Property functions are modeled only between contiguous steps. A solver finds the entire history of the reaction. It is possible to conclude the correct algorithm result from the found reaction's history. However, the last SAT version is in fact the same as the first version of the algorithm presented in the previous section.

4.4.5 Network Analysis

Realization of the mode property mechanism can follow two ways according to the time when the network reactions are computed.

- A reaction is computed at runtime when some event occurs.
- An oracle is generated at compile time. All possible reactions must be precomputed in advance. Runtime just follows the oracle when an event occurs.

The first option seems to be more straight forward but suffers with the following issue. A real-time application must be predictable, so the time, respectively WCET, of every possible reaction must be known in advance. Hence there is a need to analyze the network at deploy time and compute WCET of every reaction which can arise at runtime. The computed times are used by the application schedulability analysis. Once the application is verified, it can safely compute the network reaction at runtime when an event occurs. After each reaction a reconfiguration can be made according to the current property state.

The advantage of the runtime computation is that the only extra memory required at runtime is a representation of properties and functions. The disadvantage is significant overhead of the mode change mechanism at runtime. There is even a case where the application is correctly defined but cannot be compiled because of the overhead caused by long network reaction.

The second option uses similar network analysis at deploy time, but the purpose is different. An oracle instead of computed reaction times is a result of the analysis. The oracle is a structure which enables simulation of the network reaction at runtime in a constant time. When an event occurs the oracle is queried to return a property state after the network stabilization. Property values are adjusted according to it and a reconfiguration can be pressed.

Advantage of the oracle approach is that the time needed for the network reaction at runtime is always constant and highly predictable, so the mechanism overhead is much smaller than the overhead in the first approach. A problem could arise with runtime memory limits. The oracle must store all accessible stable property states. The number of the states can be, in the worst case, exponential with respect to the number of properties. However, the number of the states in the oracle is in an average case much smaller, as shown on a case study in Section 9.3.3.

A limiting constraint on the mechanism goes with the network analysis. Working with property domains is part of the analysis. In order to finish the analysis in finite time, the domains have to be finite. Therefore, properties are restricted to support only boolean data type and user-defined enumerations.

4.4.6 Property State Machine

The mentioned oracle can be naturally constructed as a finite-state automaton composed of stable property states. A transition between states represents a network reaction on a specific event. Assume that the automaton is in a particular property state and an event occurs. It finds a transition characterized by the event directing from the current property state. The target property state is the one which would be active if the network reacted on the event. The target state becomes current and the oracle returns it on output. Let us call this structure a *property state machine* (PSM). An example of a simple property state machine is presented in Figure 4.7.

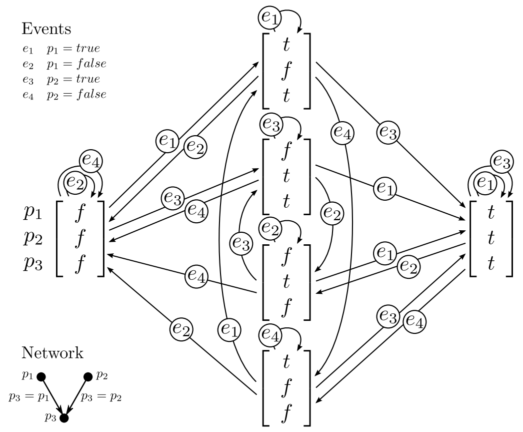


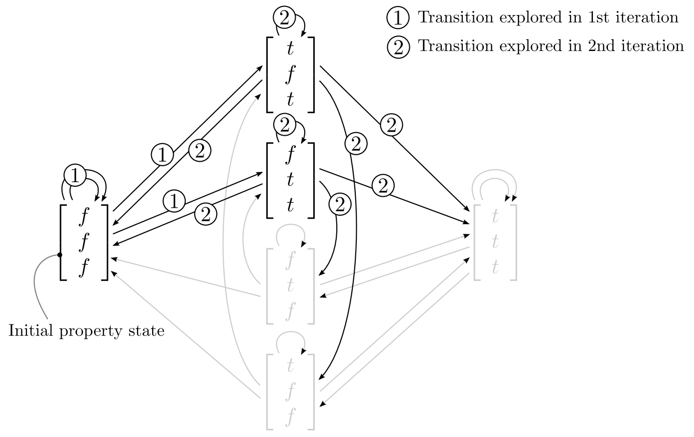
Figure 4.7: Simple property state machine

The simplest way how to find a property state machine is through the use of the following algorithm, which is called a PSM-1 algorithm. The algorithm uses BFS (or DFS) to explore all accessible stable property states forming the output property state machine. First of all, an arbitrary (initial) stable property state have to be found. The algorithm starts from it. When the initial property state is found, the exploration starts. All possible transitions from the initial property state are discovered by computing the network reaction (using the react algorithm for different events). Targets of the transitions – stable property states – are added to the output property state machine and the exploration continues with them until there are no unexplored property states. Figure 4.8 show a situation after the second iteration.

The PSM-1 algorithm has unacceptable time complexity. In a nutshell it is

$$\#\ stable\ states\ \cdot\ \#\ events\ \cdot\ T_{react}$$

where T_{react} is time complexity of the react algorithm. Unfortunately the react



1 Transition explored in 1st iteration
2 Transition explored in 2nd iteration
Initial property state

Figure 4.8: Situation during PSM-1 algorithm execution

algorithm time complexity is, in the worst case, exponential. However, let us try to reduce number of calling this subroutine.

It can be observed that the PSM-1 algorithm calls the react algorithm many times for the similar input. The react algorithm takes an event and a current property state on input. However, the algorithm does not need to know the whole property state but only a limited set of values in dependence on the input event. The react algorithm uses a set of property functions and just some of their input properties have to be necessarily known. A function's input property has to be known only if it has not been modified to the time the function is being simulated. It means that a set of necessary properties for the react algorithm and an event, let us call them *reactive* properties, is quite limited. The number of reactive properties depends on the network structure – on the number of used functions and their arity – but it is always smaller than total number of properties in the application.

Modification of the PSM-1 algorithm executes the react subroutine on the smallest necessary inputs, thus it reduces the number of calling this subroutine. It generates transition templates which can be extended to specific transitions later. The extension is done in a second phase when the initial templates are matched together until complete property states and transitions formed into a property state machine are found. The modification is called a PSM-2 algorithm. Worst case time complexity of the PSM-2 algorithm is still exponential but there is an optimization in reducing the number of called react subroutines. It is

$$\#\,events \cdot \#\,stable\ states\ on\ reactive\ properties\ (in\ dependence\ on\ event) \cdot T_{react}$$

$$+$$

$$T_{second\ phase}$$

where $T_{second\ phase}$ is time complexity of the second phase which is polynomial to the number of initial transition templates (as shown in Section 6.2.10).

4.4.7 Runtime Architecture

This section proposes an architecture of the mechanism at runtime, i.e. software parts that are automatically generated in deploy phase and integrated to the deployed application in order to implement the mode change mechanism.

The architecture is logically composed of three parts. The first part gathers values of all properties that can be directly set by the source code of components. The components have to be connected with the first architecture part in order to realize the value's transfer.

The second part integrates a simulation of network reactions. It can be either the computation at runtime using a model of the network or the computation using an oracle, both approaches described in Section 4.4.5. The approaches can be switched with respect to an application configuration or the result of an analysis determining which one is more convenient for the situation.

The third part implements a mode change protocol and handles enabling and disabling components and connections at the right time, e.g. by calling the system API. Different mode change protocols can be integrated.

It is convenient to design the second part as an extra task. A component modifying a property creates an event which is dispatched by the second architecture part and the simulation is done in the context of it. The reason is an easier WCET analysis of the mechanism overhead. The overhead for the component's task is constant. The hard work is delegated to the extra task where it can be analyzed without any dependence on common component's tasks.

4.4.8 Goals Revisited

The property based mechanism fulfills all requirements laid out in Section 4.2. The requirement (R1) – components has to remain independent and re-usable – and the requirement (R2) – the existing concept cannot be polluted – are both satisfied with the use of properties and property functions described in Sections 4.4.1 and 4.4.3. The requirement (R3)– overhead at runtime must be predictable – is satisfied with the methodology of network analysis described in Section 4.4.5 and with the preferred approach which uses oracle construction described in Section 4.4.6.

Chapter 5

Mode Change Mechanism

This chapter extends some relevant ideas introduced in Chapter 4. The design concepts of the property based mode change mechanism, i.e. properties, property functions and the property function language, are described in a formal way.

5.1 Component Mode

As described in Section 4.1, a component defines a set of component modes in dependence on its functionality and mentioned behavior. One mode of a composite component defines which subcomponents and connections are enabled and which are disabled, an example is shown in Figure 5.1. One mode of an active component defines values of real-time attributes of the component's task.

Definition 5.1. Assume a component A. If the component A is

- composite and not active, then a *component mode* M_A is a tuple $(\mathcal{S}^e, C^e)$
- primitive and active, then a *component mode* M_A is a tuple (R)
- composite and active, then a *component mode* M_A is a tuple $(\mathcal{S}^e, C^e, R)$

where

- $\mathcal{S}^e$ is a set of subcomponents contained in A which are *enabled* in the mode M_A. Other subcomponents are *disabled.* This set is called a *subcomponent enable list.* A subcomponent enable list of a component mode M is denoted $M(\mathcal{S}^e)$.
- C^e is a set of connections contained in A which are *enabled* in the mode M_A. Other connections are *disabled.* This set is called a *connection enable list.* A connection enable list of a component mode M is denoted $M(C^e)$.
- $R = (D, T, C, P)$ is a definition of real-time attributes of the component's task. This set is called a *component attribute list.* A component attribute list of a component mode M is denoted $M(R)$.

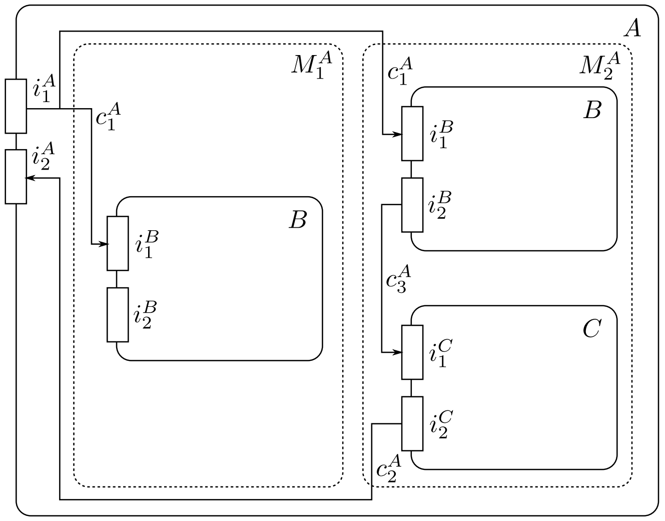


Figure 5.1: Example of a single composite component with two modes

Definition 5.2. Assume a component A. A set of all component modes defined in the context of component A is called a *component mode set* and it is denoted $\mathcal{M}_A$.

The decision which mode is active and which is inactive is based on a mode condition – a function taking (yet unspecified) variables on input and computing a mode on output. The output value corresponds to the active component mode. The mode change mechanism specifies which suitable entities stand for the input variables. It is described in the next section.

Definition 5.3. Assume a component A with a component mode set $\mathcal{M}_A$. The *mode condition* Γ_A on variables $c_1, \dots, c_n$ is a function

$$\Gamma_A : \mathcal{D}(c_1) \times \dots \times \mathcal{D}(c_n) \mapsto \mathcal{M}_A$$

The original structural requirements defined in Section 3.2 are no longer valid. They are replaced by new requirements considering component modes.

Definition 5.4. An application $\mathbf{A}$ is well defined if and only if it holds the following *(meta-model) structural requirements* for an arbitrary but fixed active (not active) composite component A contained in the application $\mathbf{A}$ and an arbitrary but fixed component mode $M = (\mathcal{S}^e, C^e, R)$ ($M = (\mathcal{S}^e, C^e)$) defined on the component A.

(1) An arbitrary require interface of an arbitrary enabled subcomponent (in $\mathcal{S}^e$) has to be connected to exactly one compatible provide interface of an enabled subcomponent (in $\mathcal{S}^e$) or to exactly one compatible require interface of the component A.

(2) An arbitrary provide interface of the component A has to be connected to exactly one provide interface of an enabled subcomponent (in $\mathcal{S}^e$).

(3) An arbitrary enabled connection (in C^e) has to connect only interfaces defined on enabled subcomponents (in $\mathcal{S}^e$) or on the component A.

The corollary implied from the structural requirements is similar to the corollary described in Section 3.2 but it is extended to consider component modes. It means that the theorem is valid in an arbitrary application's configuration and a thread of execution cannot reach any disabled component.

The structural requirement (3) is not necessary. It is included in order to disable all connections which cannot be used, thus to preserve lucidity of the designed component model.

5.2 Properties and Property Functions

The property based mode change mechanism introduced in Chapter 4 has two main intentions – to provide an application-wide communication for mode change handling and to model a mode condition for each component which needs it. It is realized using properties and property functions.

5.2.1 Component and Application Properties

A set of common properties for each component is defined. The properties are variables with finite domains defined in the context of the component.

Definition 5.5. Assume a component A. A set of variables $p_1^A, \dots, p_m^A$ defined in the context of component A is called a *component property set*. Let us denote it $\mathcal{P}_A$. Each variable is called a *component property*.

Definition 5.6. Let p be a component property. A finite set of values $\mathcal{D}(p)$ is the *domain* of the property p.

It is also possible to define "domain" of an arbitrary set of properties as Cartesian product of involved property domains. This construction is helpful for the following definitions in order to keep them simple.

Definition 5.7. Let $\mathcal{P} = \{p_1, \dots, p_n\}$ be a set of arbitrary component properties. Then

$$\mathcal{D}(\mathcal{P}) = \mathcal{D}(p_1) \times \cdots \times \mathcal{D}(p_n)$$

An application is just a bunch of components composed on different levels. Each component defines component properties. It is useful to work with all properties defined in the application together. This application-wide union of component properties is simply called a property set, an example is shown in Figure 5.2.

Definition 5.8. Assume an application $\mathbf{A}$. Let $A_1, A_2, \dots, A_k$ be all components contained in the application $\mathbf{A}$. For each $i = 1, \dots, k$ the component A_i defines a component property set $\mathcal{P}_{A_i} = \{p_1^{A_i}, \dots, p_{m_i}^{A_i}\}$. A set of variables

$$\mathcal{P}_{\mathbf{A}} = \{p_1^{A_1}, \dots, p_{m_1}^{A_1}, p_1^{A_2}, \dots, p_{m_2}^{A_2}, \dots, p_1^{A_k}, \dots, p_{m_k}^{A_k}\}$$

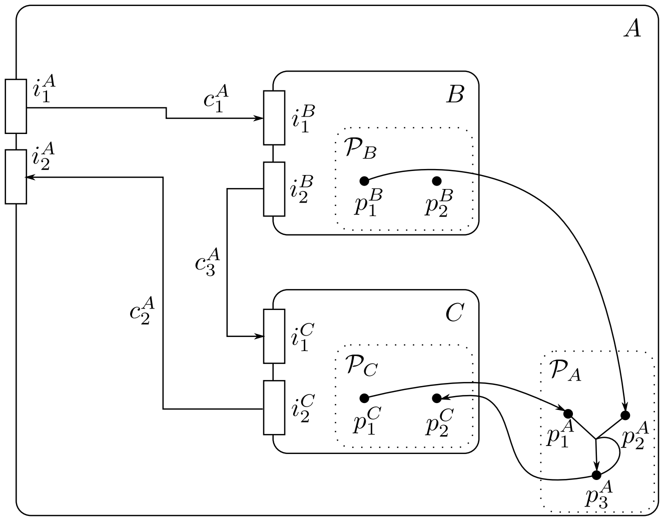


Figure 5.2: Example of component properties and functions defined on a composite component with two subcomponents

is called an *(application) property set* of the application $\mathbf{A}$. Each variable in a property set is called an *(application) property*.

5.2.2 Property Network

A communication across the application for the purpose of mode change handling is realized through the use of property functions composed into a property network. A property function is an ordinary function taking values of input properties and computing a value of an output property. It works in the context of the network which reacts when an event occurs. The events are more closely described in Section 5.2.5. An example of a property network can be seen in Figure 5.3.

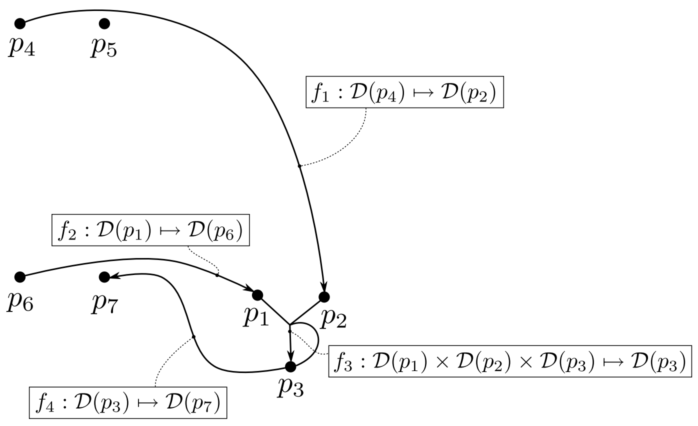
Definition 5.9. Let $\mathbf{A}$ be an application with a property set $\mathcal{P}_{\mathbf{A}} = \{p_1, \ldots, p_n\}$. Let F be a set of functions

$$F \subseteq \{f : \mathcal{D}(I) \mapsto \mathcal{D}(p) \mid \forall I \subseteq \mathcal{P}_{\mathbf{A}}, \forall p \in \mathcal{P}_{\mathbf{A}}\}$$

A pair $N_{\mathbf{A}} = (\mathcal{P}_{\mathbf{A}}, F)$ is called a *property network* of the application $\mathbf{A}$ and functions in F are called *property functions*. A set of property functions in the property network $N_{\mathbf{A}}$ is denoted $N_{\mathbf{A}}(F)$.

Definition 5.10. Assume an application $\mathbf{A}$ with a property set $\mathcal{P}_{\mathbf{A}}$. Let $I \subseteq \mathcal{P}_{\mathbf{A}}$ be a set of properties, $p \in \mathcal{P}_{\mathbf{A}}$ be a property and $f : \mathcal{D}(I) \mapsto \mathcal{D}(p)$ be a property function. Then

- I is called an *input property set* of the property function f and it is denoted $i(f)$.
- p is called an *output property* of the property function f and it is denoted $o(f)$.



p_4
p_5
$f_1 : \mathcal{D}(p_4) \mapsto \mathcal{D}(p_2)$
$f_2 : \mathcal{D}(p_1) \mapsto \mathcal{D}(p_6)$
p_6
p_7
p_1
p_2
$f_3 : \mathcal{D}(p_1) \times \mathcal{D}(p_2) \times \mathcal{D}(p_3) \mapsto \mathcal{D}(p_3)$
p_3
$f_4 : \mathcal{D}(p_3) \mapsto \mathcal{D}(p_7)$

Figure 5.3: Example of a property network corresponding to Figure 5.2

Definition 5.11. Let $\mathbf{A}$ be an application with a property set $\mathcal{P}_\mathbf{A}$ and a property network $N_\mathbf{A}$. Let p_k and p_l be properties in $\mathcal{P}_\mathbf{A}$. The property p_l is *dependent* on the property p_k if and only if there exists an oriented path (created by property functions) in the network $N_\mathbf{A}$ from p_k to p_l.

5.2.3 Component Mode Condition

The mode change mechanism models a component mode like one of the component properties. This construction has an advantage that the component mode can be used as input for computing a value of some other property or even the component mode itself.

This construction also specifies which entities stand for the component mode condition and its variables (5.3). The condition is an arbitrary property function with this special property on output. Input properties of all such functions stand for the condition variables.

Definition 5.12. Assume a composite component A with a component property set $\mathcal{P}_A$ and a component mode set $\mathcal{M}_A = \{M_1^A, \ldots, M_m^A\}$. The component property set $\mathcal{P}_A$ is *well defined* if and only if $\mathcal{P}_A$ contains a property p_{mode} with the domain $\mathcal{D}(p_{mode}) = \{M_1^A, \ldots, M_m^A\}$. The property p_{mode} is *corresponding* to each of the modes $M_1^A, \ldots, M_m^A$.

Theorem 5.13. *Assume an application* $\mathbf{A}$ *with a property network* $N_\mathbf{A}$. *Assume a composite component* A *contained in* $\mathbf{A}$ *with a well defined component property set* $\mathcal{P}_A$ *and a component mode set* $\mathcal{M}_A$. *An arbitrary property function* $f \in N_\mathbf{A}(F) : o(f) = p_{mode}$ *is the mode condition of the component* A.

5.2.4 Network Functionality

The react algorithm (introduced in Section 4.4.3) is a routine which represents a reaction of the property network on an event. The event assigns a value to a property, called an initial property. The algorithm also defines two sets of properties in dependence on the initial property. Reactive properties are such properties whose values have to be necessarily known in order to properly represent the network reaction. Affected properties are such properties which have been modified during the network reaction. Example of the two sets of properties is presented in Figure 5.4 (a).

Definition 5.14. Assume an application $\mathbf{A}$ with a property set $\mathcal{P}_{\mathbf{A}} = \{p_1, \ldots, p_n\}$ and a property network $N_{\mathbf{A}}$. Let us define an algorithm *react* which modifies an evaluation of properties in $\mathcal{P}_{\mathbf{A}}$. The algorithm takes two arguments – an arbitrary property $p_i \in \mathcal{P}_{\mathbf{A}}$, called an *initial property*, and an arbitrary initial property value $\alpha \in \mathcal{D}(p_i)$.

```
procedure react($p_i$,$\alpha$)
    create queue $Q$
    $p_i$ := $\alpha$                                  (1)
    foreach $f$ from $N_{\mathbf{A}}$:  $p_i$ in $i(f)$
       enqueue $f$ into $Q$
    while $Q$ is not empty
       dequeue $f$ from $Q$
       let $p_o$ be $o(f)$
       let $\{p_{i_1}, \ldots, p_{i_k} \mid 1 \leq k \leq n\}$ be $i(f)$    (2)
       $p_o$ := $f(p_{i_1}, \ldots, p_{i_k})$                  (3)
       foreach $g$ from $N_{\mathbf{A}}$:  $p_o$ in $i(g)$
          enqueue $g$ into $Q$
```

Definition 5.15. Assume an application $\mathbf{A}$ with a property set $\mathcal{P}_{\mathbf{A}} = \{p_1, \ldots, p_n\}$ and a property network $N_{\mathbf{A}}$. Let p_i be an arbitrary but fixed property in $\mathcal{P}_{\mathbf{A}}$ and α a value in its domain. Let us simulate execution of *react*(p_i, α) (5.14) and construct two sets of properties – *affected* properties $Af_{\mathbf{A}}(p_i)$ and *reactive* properties $Re_{\mathbf{A}}(p_i)$ – according to the following rules which are applied on the denoted lines of pseudocode.

(1) Add p_i to $Af_{\mathbf{A}}(p_i)$.

(2) Add all properties from $i(f)$ which are not currently in $Af_{\mathbf{A}}(p_i)$ to $Re_{\mathbf{A}}(p_i)$.

(3) Add p_o to $Af_{\mathbf{A}}(p_i)$.

Considering a particular network reaction, each property can be evaluated with a number, called order, that describes in which step of the react algorithm (5.14) is the property modified for the last time. The larger order of the property means its later modification by the react algorithm. An example of an order evaluation is shown in Figure 5.4 (b).

Properties on a cycle or properties dependent on them are modified repeatedly (because of infinite nature of the react algorithm) and thus their order is evaluated to ∞. For other properties, order corresponds to the longest possible path from the initial property.

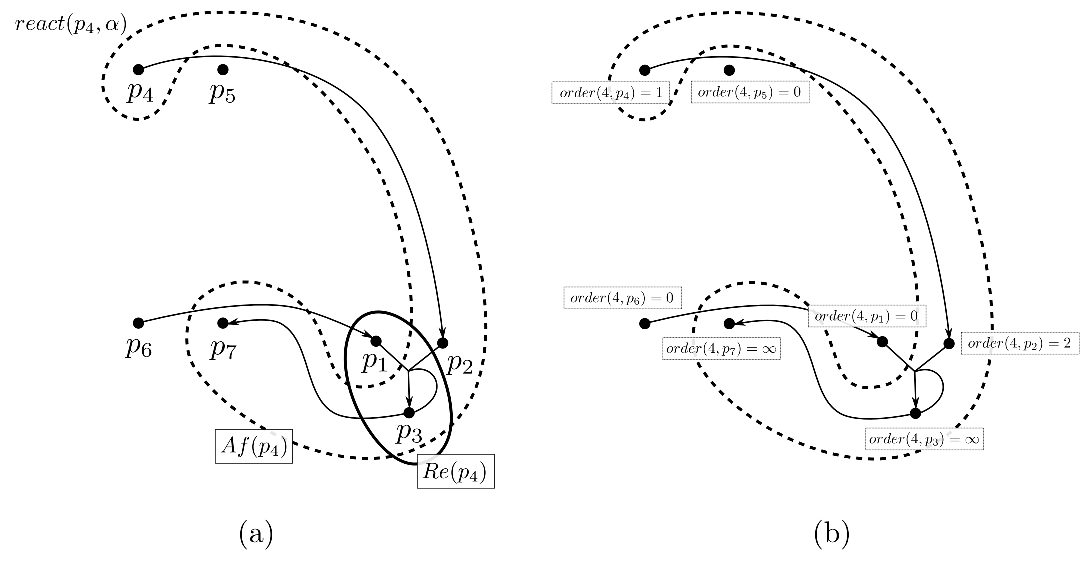


Figure 5.4: Example of affected and reactive properties and the function *order*

Definition 5.16. Assume an application $\mathbf{A}$ with a property set $\mathcal{P}_{\mathbf{A}} = \{p_1, \ldots, p_n\}$ and a property network $N_{\mathbf{A}}$. Let $p_i \in \mathcal{P}_{\mathbf{A}}$ be an arbitrary but fixed property, $Af_{\mathbf{A}}(p_i)$ a set of affected properties and $p_j \in \mathcal{P}_{\mathbf{A}}$ an arbitrary property. Let us define a function $order(i, p_j)$ by the following rule.

$$order(i, p_j) = \begin{cases} 0 & \ldots \quad p_j \notin Af_{\mathbf{A}}(p_i) \\ \infty & \ldots \quad p_j \in Af_{\mathbf{A}}(p_i) \wedge \\ & \qquad p_j \text{ in cycle or dependent on cycle} \\ (\text{max path } p_i \text{ to } p_j) + 1 & \ldots \quad \textit{else} \end{cases}$$

5.2.5 Network Input

The way how to pass a particular value to the property network has not been discussed yet. It is natural that the application should be able to set a value of a property defined on a component which contains a source code. This option can be part of component API. But if a component has no source code there is no way how to directly set a value of a property defined on it. It gives a division to direct and indirect properties.

Definition 5.17. Assume a component A with a component property set $\mathcal{P}_A$. Each component property in $\mathcal{P}_A$ is

- *direct* if and only if the component A contains a source code. Values of these properties can be set directly from the component source code through the component API.

- *indirect* if and only if the component A does not contain any source code. Values of these properties cannot be set directly but only by a property function during the network reaction.

Assigning an arbitrary value to a direct property is an event which can arise in the mode change mechanism. There are no other events possible in the mechanism. Therefore, direct properties and its domains define all possible events.

Definition 5.18. Assume an application $\mathbf{A}$ with a property set $\mathcal{P}_{\mathbf{A}}$. Let $\mathcal{P}_{\mathbf{A}}^{Di} \subseteq \mathcal{P}_{\mathbf{A}}$ be a set of all direct properties. Then a set

$$E_{\mathbf{A}} = \{(p_i, \alpha) \mid \forall p_i \in \mathcal{P}_{\mathbf{A}}^{Di} : \forall \alpha \in \mathcal{D}(p_i)\}$$

is called an *event set* of the application $\mathbf{A}$.

5.3 Property Function Language

A language for property function behavior description is introduced in Chapter 4. It is a domain specific language newly created for the purpose of the thesis and introduced to meta-model.

Because of the mentioned usage and property domains limited to boolean data types and user-defined enumerations, the language can be simple. In a nutshell the language is a set of statements producing an output value on the basis of satisfied boolean expression. A boolean expression works with input variables, constants, simple relational operators and standard boolean operators.

5.3.1 Syntax

Syntax of the property function language is presented in Figure 5.5. The figure uses Extended Backus–Naur Form (EBNF) defined by ISO [16]. The syntax description contains a complete language grammar except for the last two lines where notation is shortened in order to save space. The first part describes statements and expressions. The second part specifies lexical elements like variables and constants.

5.3.2 Semantics

Semantics of one property function is simple. The set of statements (`stmt`) is inspected in sequence and boolean expressions (`expr`) are evaluated according to input values. When a boolean expression is evaluated to *true*, the statement takes effect and the value of expression written on the right side is returned from the function. Then the execution ends. It means that the result is the value of the right side of the first matched statement.

Boolean expressions can use only input variables declared in the function header (`head`). Evaluation of boolean expression is the same as in the "ordinary" programming languages like Java or C++. First, relational operators (`relexpr`) are resolved. Values are compared and the relational operator is evaluated to *true* or *false*. A relational operator is the only place where variables with non-boolean data type can be used. When relational operators are resolved, boolean operators are applied according to the expression structure and a value of boolean expression is computed.

```
     func  =  head, ‘{’, { stmt }, ‘}’ ;
     head  =  id, ‘(’, { param }, ‘)’ ;
    param  =  id
     stmt  =  expr, ‘=>’, expr ‘;’ ;
     expr  =  expr, ‘||’, andexpr | andexpr ;
  andexpr  =  andexpr, ‘&&’ notexpr | notexpr ;
  notexpr  =  ‘!’, parexpr | parexpr ;
  parexpr  =  ‘(’, expr, ‘)’ | relexpr ;
  relexpr  =  atom, rel, atom | atom ;
     atom  =  const | param ;

      rel  =  ‘=’ | ‘!=’ ;
       id  =  letter, { letter | digit | ‘_’ } ;
    const  =  boolconst | strconst ;
boolconst  =  ‘false’ | ‘true’ ;
 strconst  =  ‘"’, { letter | digit | ‘_’ }, ‘"’ ;
   letter  =  ‘a’ | ... | ‘z’ | ‘A’ | ... | ‘Z’ ;
    digit  =  ‘0’ | ‘1’ | ... | ‘9’ ;
```

Figure 5.5: Syntax of the property function language

The language does not have a strong typing. This approach is more suitable for the mentioned usage as a simple language for creating function templates. Data types of parameters are resolved when the property function is applied on particular properties.

Chapter 6

Oracle Construction

As described in Section 4.4.5, the rare concept of the property based mode change mechanism is not suitable for a low-level real-time application because of the predictability problem. It must be supplemented with an analysis which guarantees fulfilling of real-time requirements. One of the possible solutions is to introduce an oracle. It is a structure generated in deploy phase which helps at runtime to reduce overhead of the mode change mechanism. The network reaction that uses the oracle has the same inputs and the same outputs as the network reaction defined in Section 4.4.3 but time to compute the reaction is constant thus predictable. The first part of this chapter describes a construction of such an oracle.

Output of the oracle is an evaluation of properties – a property state. Each such state can be associated with an actual configuration of the application, i.e. which components and connections are enabled and which values of real-time attributes are applied. The second part of this chapter introduces a mathematical structure describing the configurations and constructs an algorithm which finds a configuration associated to a particular property state.

6.1 Property State Machine

The oracle is designed as a property state machine, i.e a finite-state automaton representing all possible network reactions and suggesting stable property states. It operates with property states. One property state is an evaluation of each property in the application.

Definition 6.1. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$. A *property state* A is a vector of values $\alpha_1, \ldots, \alpha_n$ corresponding to the properties $p_1, \ldots, p_n$.

$$\alpha_i \in \mathcal{D}(p_i),\ \forall i = 1, \ldots, n$$

There are specific property states contained in the property state machine. They exist at the end of a network reaction when the network is stabilized. Their values are not spontaneously changed on the basis of some property function. Therefore, they are called stable property states.

A stable property state can be recognized by the following analysis. For each property, there had to be a network reaction that set the analyzed property to its current value. It can be denoted as transitive supporting – the analyzed property is transitively supported by the initial property of the reaction. Moreover, there is a path from the initial property to the analyzed property on which all properties are also transitively supported by the same initial property. Particular steps of the path are realized by supporting functions. A supporting function represents a function which modified the analyzed property for the last time.

Definition 6.2. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$, a property network $\mathcal{N} = (\mathcal{P}, F)$, an arbitrary but fixed direct property $p_k \in \mathcal{P}$ and an arbitrary property $p_i \in \mathcal{P}$. Let $F_i \subseteq F$ be a set of property functions with the property p_i as output.

$$F_i = \{f \in F \mid o(f) = p_i\}$$

Let $\mathcal{P}_i$ be a set of input properties of all functions in F_i.

$$\mathcal{P}_i = \{p \in \mathcal{P} \mid \exists f \in F_i : \ p \in i(f)\}$$

Finally, let $S_i^k \subseteq F_i$ be a set of functions for which there exists an input property p with the maximal positive $order(k, p)$ among all properties in $\mathcal{P}_i$.

$$S_i^k = \{f \in F_i \mid \exists p \in \mathcal{P}_i : \ p \in i(f) \ \wedge \ order(k,p) > 0 \ \wedge \ \max_{\mathcal{P}_i}(order(k,p))\}$$

Then S_i^k is a *set of k-supporting functions* of the property p_i. An example of such a set of functions is shown in Figure 6.1 (a).

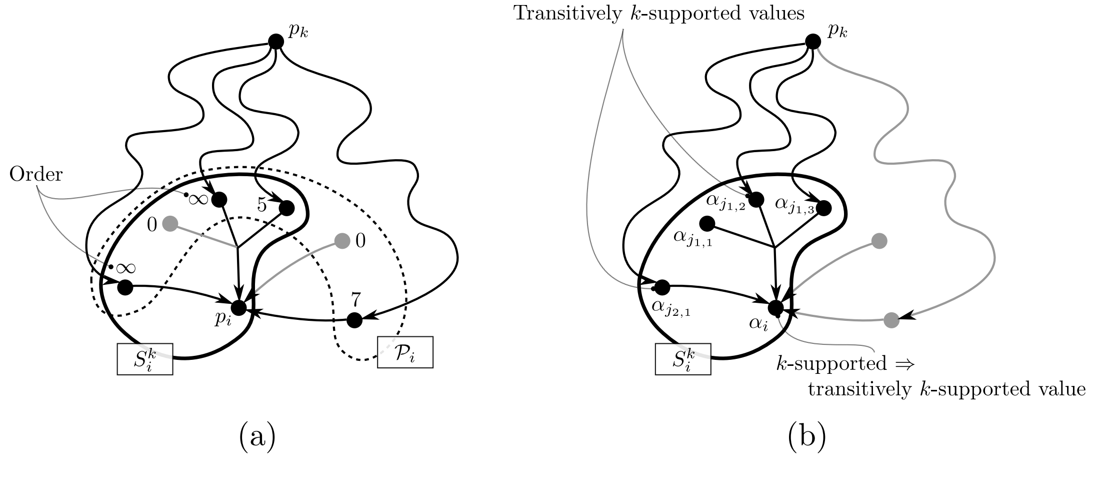


Figure 6.1: Example of k-supporting functions and transitive k-supporting

Definition 6.3. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$, a property state $A = (\alpha_1, \ldots, \alpha_n)$ defined on $\mathcal{P}$, an arbitrary but fixed direct property $p_k \in \mathcal{P}$ and an arbitrary property $p_i \in \mathcal{P}$. Let S_i^k be a set of k-supporting functions of the property p_i. A property value α_i is *k-supported* if and only if all k-supporting functions of the property p_i compute the value α_i.

$$\forall f \in S_i^k \ (f : \mathcal{D}(p_{j_1}, \dots p_{j_m}) \mapsto \mathcal{D}(p_i)) : \ f(\alpha_{j_1}, \dots, \alpha_{j_m}) = \alpha_i$$

Definition 6.4. Assume an application with a property set $\mathcal{P} = \{p_1, \dots, p_n\}$, a property state $A = (\alpha_1, \dots, \alpha_n)$ defined on $\mathcal{P}$, an arbitrary but fixed direct property $p_k \in \mathcal{P}$ and an arbitrary property $p_i \in \mathcal{P}$. A property value α_i is *transitively k-supported* if and only if

- $p_i = p_k$, or
- α_i is k-supported and for each k-supporting function f of the property p_i there exists $p_j \in i(f)$, $p_j \neq p_i$ that its corresponding value $\alpha_j \in A$ is also transitively k-supported. An example of such a situation is shown in Figure 6.1 (b).

Definition 6.5. Assume an application with a property set $\mathcal{P}$ and a property state A defined on $\mathcal{P}$. Let $\mathcal{P}^{Di} \subseteq \mathcal{P}$ be a set of all direct properties. The property state A is *stable* if and only if

$$\forall \alpha \in A : \ (\exists p_k \in \mathcal{P}^{Di} : \ \alpha \text{ is transitively } k\text{-supported})$$

A process of the network reaction is in a property state machine represented by a transition from one stable property state to another. Each transition is labeled by a transition characteristic which represents the event that invoked the reaction.

Definition 6.6. Assume an application with a property set $\mathcal{P} = \{p_1, \dots, p_n\}$, a property network N and a stable property state A. Let p_i be a direct property in $\mathcal{P}$ and γ be an arbitrary property value in $\mathcal{D}(p_i)$. Let us apply the algorithm *react*(p_i, γ) on the property state A creating a new property state B.

A pair of property states (A, B) is called a *stable transition.* A pair (p_i, γ) is called a *stable transition characteristic.*

Theorem 6.7. *Let (A, B) be a stable transition. Then B is a stable property state.*

Proof. Assume $A = (\alpha_1, \dots, \alpha_n)$ and $B = (\beta_1, \dots, \beta_n)$ defined on property set $\mathcal{P} = \{p_1, \dots, p_n\}$. Let us remind that the react algorithm affects all properties which are dependent on the initial property p_i. B is created by a modification of A during the reaction. For each value $\beta_j \in B$ there are two possibilities.

- The property p_j is not affected during the reaction. But also no property which is p_j dependent on can be affected during the reaction (otherwise p_j would be affected). It means that β_j is transitively k-supported in B with the same k as the corresponding value α_j in A.
- The property p_j is affected during the reaction. The react algorithm affects properties in a way that β_j is transitively i-supported in B where i is the index of the initial property.

□

Finally, the desirable property state machine is a state machine constructed only from stable property states and stable transitions (assured in 6.8). A transition is invoked by an event which is input of the state machine. The second condition (described in 6.9) requires that each contained property state has outbound transitions defined for all possible events. Reason for this is very simple. The state machine must be in an arbitrary property state able to continue no matter which input comes. An example of a well defined stable property state machine is presented in Figure 6.2.

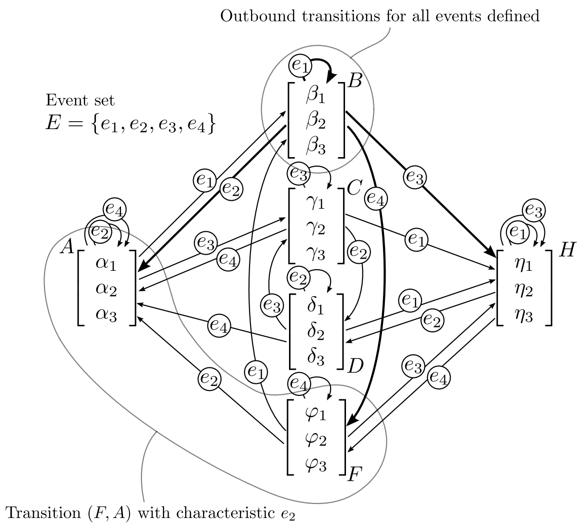

Outbound transitions for all events defined
Event set
$E = \{e_1, e_2, e_3, e_4\}$
A
B
C
D
F
H
α_1 α_2 α_3
β_1 β_2 β_3
γ_1 γ_2 γ_3
δ_1 δ_2 δ_3
φ_1 φ_2 φ_3
η_1 η_2 η_3
e_1 e_2 e_3 e_4
Transition (F, A) with characteristic e_2

Figure 6.2: Well defined stable property state machine

Definition 6.8. A *stable property state machine* is an oriented graph (S, T) where nodes (S) correspond to stable property states and edges (T) correspond to stable transitions.

Definition 6.9. Assume an application with an event set E (5.18) and a stable property state machine $PSM = (S, T)$. PSM is *well defined* if and only if $S \neq \emptyset$ and for each property state $A \in S$

$$\forall e \in E : (\exists (A, B) \in T : characteristic(e, (A, B)))$$

6.2 PSM Algorithm

Two algorithms for oracle construction has been described in Section 4.4.6. This section presents the algorithm PSM-2 in more detail.

Input of the algorithm is a set of properties formed into a property network. Output of the algorithm is well defined stable property state machine build on the property set according to the previous definitions.

6.2.1 Basic Idea

The algorithm works with "incomplete" property states. This incompleteness is based on temporary property values λ representing that the real value is not known yet. Incomplete property states are combined with each other and λ values are gradually eliminated and replaced with the real values. However, the incomplete property states keep an invariant through the entire algorithm execution. The invariant guarantees that the completed property states are formed into a well defined stable property state machine.

6.2.2 PSM State

A property state possibly containing a λ value is called a PSM state. If a PSM state contains a λ value it is called open. It is closed when the PSM state is completed and all λ values are eliminated. In the completed form it is equivalent to a property state.

Definition 6.10. Let symbol λ denote special value that is not a member of the domain of any property. This symbol represents that the real value is not known.

Definition 6.11. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$. A *PSM state* A is a vector of values $\alpha_1, \ldots, \alpha_n$.

$$\alpha_i \in \mathcal{D}(p_i) \cup \{\lambda\},\ \forall i = 1, \ldots, n$$

Definition 6.12. Assume a PSM state $A = (\alpha_1, \ldots, \alpha_n)$. The PSM state A is *open* if and only if $\exists i : \alpha_i = \lambda$. Otherwise the PSM state A is *closed.*

Theorem 6.13. *A closed PSM state is equivalent to a property state.*

Proof. Assume a closed PSM state $A^{PSM} = (\alpha_1, \ldots, \alpha_n), \forall i : \alpha_i \neq \lambda$. It has the same structure as a property state $A^{PS} = (\alpha_1, \ldots, \alpha_n)$ which is the demanded property state. $\square$

It is possible to define stability of a PSM state in a similar way as stability of property state. λ values are the only problem which must be handled. The main idea is following. If a function has a λ value on input, the input is incomplete and the function can possibly support its output value when it is supplemented with right values. A decision if the function supports its output value is made when all λ values on input are eliminated and replaced with real values.

Definition 6.14. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$, a PSM state $A = (\alpha_1, \ldots, \alpha_n)$ defined on $\mathcal{P}$, an arbitrary but fixed direct property $p_k \in \mathcal{P}$ and an arbitrary property $p_i \in \mathcal{P}$. Let S_i^k be a set of k-supporting functions of the property p_i. A property value α_i is *k-supported* if and only if each k-supporting function of the property p_i has either λ value on input or computes the value α_i.

$$\forall f \in S_i^k\ (f : \mathcal{D}(p_{j_1}, \ldots p_{j_m}) \mapsto \mathcal{D}(p_i)) :\ (\lambda \in \{\alpha_{j_1}, \ldots, \alpha_{j_m}\}\ \vee\ f(\alpha_{j_1}, \ldots, \alpha_{j_m}) = \alpha_i)$$

Definition 6.15. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$, a PSM state $A = (\alpha_1, \ldots, \alpha_n)$ defined on $\mathcal{P}$, an arbitrary but fixed direct property $p_k \in \mathcal{P}$ and an arbitrary property $p_i \in \mathcal{P}$. A property value α_i is *transitively k-supported* if and only if

- $p_i = p_k$, or
- α_i is k-supported and for each k-supporting function f of the property p_i there exists $p_j \in i(f)$, $p_j \neq p_i$ that its corresponding value $\alpha_j \in A$ is also transitively k-supported.

Definition 6.16. Assume an application with a property set $\mathcal{P}$ and a PSM state A defined on $\mathcal{P}$. Let $\mathcal{P}^{Di} \subseteq \mathcal{P}$ be a set of all direct properties. The PSM state A is *stable* if and only if

$$\forall \alpha \in A : \ (\exists p_k \in \mathcal{P}^{Di} : \ \alpha \text{ is transitively } k\text{-supported})$$

Theorem 6.17. *A closed stable PSM state is equivalent to a stable property state.*

Proof. A closed PSM state A^{PSM} is according to 6.13 equivalent to a property state A^{PS}. The PSM state A^{PSM} is stable and does not contain any λ value. It means that the same supporting functions are satisfied in A^{PS} and the property state A^{PS} matches the definition 6.5. □

6.2.3 PSM Transition

PSM states create stable transitions similar to transitions defined on property states. They are called stable PSM transitions.

Definition 6.18. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$, a property network N and a stable PSM state A. Let p_i be a direct property in $\mathcal{P}$ and γ be an arbitrary property value in $\mathcal{D}(p_i)$. Assume that $\lambda \notin Re(p_i)$. Let us apply the algorithm *react(p_i,γ)* on the PSM state A creating a new PSM state B.

A pair of PSM states (A, B) is called a *stable PSM transition.* The first state is called a *source state*, the second is called a *target state.* A pair (p_i, γ) is called a *stable PSM transition characteristic.*

Theorem 6.19. *Let (A, B) be a stable PSM transition. Then B is stable PSM state.*

Proof. The proof is similar to 6.7. □

Theorem 6.20. *Let (A, B) be a stable PSM transition and A a closed PSM state. Then the PSM state B is closed.*

Proof. B is created by a modification of A during the reaction. The react algorithm does not generate any new λ values. No λ value is in A, hence no λ value is in B. □

Theorem 6.21. *Let (A, B) be a stable PSM transition and A and B closed stable PSM states. Then it is equivalent to a stable transition.*

Proof. According to 6.17 A and B are equivalent to stable property states and the definition of the stable PSM transition (A, B) matches the definition 6.6. □

A term very important for the algorithm is an initial stable PSM transition. It is a stable PSM transition which can be generated directly from a property network. This transition describes just one network reaction and nothing more. All values not involved in a reaction are evaluated to λ. Generating the initial transitions is the first phase of the algorithm. The initial transitions are passed to the second phase as input and the algorithm creates an output state machine on the basis of them.

Definition 6.22. Assume an application with a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$ and a stable PSM transition (A, B) with characteristic (p_i, β_i) where $A = (\alpha_1, \ldots, \alpha_n)$ and $B = (\beta_1, \ldots, \beta_n)$ defined on $\mathcal{P}$. (A, B) is called an *initial stable PSM transition* if and only if

$$\forall j :\ p_j \in (Re(p_i) \cup \{p_i\}) \iff \alpha_j \neq \lambda$$

6.2.4 PSM State Combination

One of the algorithm steps is combination of two compatible stable PSM states together in order to eliminate some λ values and create a new more complete stable PSM state. Combination lies in substitution of λ values in one state with values from the other state. Values in the same positions in compatible states must be either equal or one of them must be λ. Moreover, a product of a combination must be a stable PSM state again.

Definition 6.23. Assume two values α and β.

- The values are *compatible* (denoted as $\alpha \sim \beta$) if and only if $\alpha = \beta \ \vee \ \alpha = \lambda \ \vee \ \beta = \lambda$.
- The values are *left compatible* (denoted as $\alpha \succ \beta$) if and only if $\alpha = \beta \vee \beta = \lambda$.

Definition 6.24. Let us define a binary operation $\circledast$ on two compatible values α and β.

$$\alpha \circledast \beta = \begin{cases} \alpha & \ldots \quad \beta = \lambda \ \vee \ \alpha = \beta \\ \beta & \ldots \quad \alpha = \lambda \end{cases}$$

Definition 6.25. Assume two vectors $A = (\alpha_1, \ldots, \alpha_n)$ and $B = (\beta_1, \ldots, \beta_n)$.

- The vectors are *compatible* (denoted as $A \sim B$) if and only if

$$\forall i : \alpha_i \sim \beta_i$$

- The vectors are *left compatible* (denoted as $A \succ B$) if and only if

$$\forall i : \alpha_i \succ \beta_i$$

Definition 6.26. Let us define a binary operation $\circledast$ on two compatible vectors $A = (\alpha_1, \dots, \alpha_n)$ and $B = (\beta_1, \dots, \beta_n)$.

$$A \circledast B = \begin{pmatrix} \alpha_1 \circledast \beta_1 \\ \dots \\ \alpha_n \circledast \beta_n \end{pmatrix}$$

Definition 6.27. Assume two stable PSM states $A = (\alpha_1, \dots, \alpha_n)$ and $B = (\beta_1, \dots, \beta_n)$.

- The stable PSM states are *compatible* (denoted as $A \sim B$) if and only if $A \sim B$ (from the vector point of view) and $A \circledast B$ is a vector equivalent to a stable PSM state.
- The stable PSM states are *left compatible* (denoted as $A \succ B$) if and only if $A \succ B$ (from the vector point of view) and $A \circledast B$ is a vector equivalent to a stable PSM state.

Definition 6.28. Let us define a binary operation $\circledast$ on two compatible stable PSM states $A = (\alpha_1, \dots, \alpha_n)$ and $B = (\beta_1, \dots, \beta_n)$.

$$A \circledast B = \begin{pmatrix} \alpha_1 \circledast \beta_1 \\ \dots \\ \alpha_n \circledast \beta_n \end{pmatrix}$$

Aside stable PSM states, even a stable PSM transition can be combined with a stable PSM state. The combination is directed by the source state of the transition that has to be compatible with the second operand. A new stable PSM transition is a result of the operation and represents the idea "Combine the source state of the original transition with the second operand and then apply the react algorithm with the same input as the original transition".

Definition 6.29. Assume a stable PSM transition (A, B) and a stable PSM state C where $A = (\alpha_1, \dots, \alpha_n)$, $B = (\beta_1, \dots, \beta_n)$ and $C = (\gamma_1, \dots, \gamma_n)$. Let A and C be the compatible stable PSM states. Let us define a binary operation $\circledast$ as

$$C \circledast (A, B) = (A \circledast C, B')$$

where $B' = (\beta'_1, \dots, \beta'_n)$ and

$$\forall \beta'_i \in B' : \ \beta'_i = \begin{cases} \gamma_i & \beta_i = \lambda \\ \beta_i & \beta_i \neq \lambda \end{cases}$$

A combination of a stable PSM state and a stable PSM transition preserves stability. This is very important characteristic for the algorithm to be correct. This idea is proved in the following theorem.

Theorem 6.30. *Assume a stable PSM transition (A, B) and a stable PSM state C. Let A and C be the compatible stable PSM states. Then $C \circledast (A, B) = (A \circledast C, B')$ is a stable PSM transition.*

Proof. Assume a property set $\mathcal{P} = \{p_1, \ldots, p_n\}$ and PSM states $A = (\alpha_1, \ldots, \alpha_n)$, $B = (\beta_1, \ldots, \beta_n)$ and $C = (\gamma_1, \ldots, \gamma_n)$. Let us apply the react algorithm on the PSM state $A \circledast C$ with the same characteristic as the PSM transition (A, B) has. If the algorithm modifies the PSM state $A \circledast C$ to the PSM state B' then the result is a stable PSM transition according to the definition 6.18.

Let (p_i, β_i) be the transition characteristic. The react algorithm works for the same characteristics and the same values on reactive properties equally – it affects the same set of properties and assigns the same (non-λ) values. Let us denote the following sets of indices.

$$K := \{k \mid p_k \in Af(p_i)\},\ J := \{j \mid p_j \notin Af(p_i)\}$$

For an arbitrary index $k \in K$, it holds that

$$\beta_k \neq \lambda \ \wedge \ \beta'_k = \beta_k$$

For an arbitrary index $j \in J$, there are two situations.

- $\beta_j \neq \lambda \implies (\beta_j = \alpha_j = \alpha_j \circledast \gamma_j = \beta'_j)$
- $\beta_j = \lambda \implies (\alpha_j = \lambda) \implies (\gamma_j = \alpha_j \circledast \gamma_j = \beta'_j)$

□

6.2.5 Preprocessing

A reaction can never reach a connected component of the property network which is composed only from indirect properties. Therefore, each such a component has to be erased from the network. The algorithm would never create closed PSM state if such a component existed in the network.

6.2.6 First Phase

The first phase of the algorithm creates all possible initial stable PSM transitions from the input (preprocessed) property network. It is done according to the definitions 6.18 and 6.22 by the following algorithm. Output of the fist phase is stored in a graph $\mathcal{G}_0 = (\mathcal{N}_0, \mathcal{L}_0)$ whose exact meaning is described in the next section.

```
$\mathcal{N}_0$ := $\emptyset$
$\mathcal{L}_0$ := $\emptyset$
$\mathcal{G}_0$ := $(\mathcal{N}_0, \mathcal{L}_0)$
foreach $(p_i, \gamma) \in E$
  foreach $A$: stable state on $Re(p_i) \cup \{p_i\}$
    apply $react(p_i, \gamma)$ on $A$ creating $B$
    $\mathcal{N}_0$ := $\mathcal{N}_0 \cup \{A, B\}$
    $\mathcal{L}_0$ := $\mathcal{L}_0 \cup \{(A, B)\}$
```

6.2.7 Second Phase

The algorithm works in iterations. Input of an iteration is a graph (stable PSM states and transitions) which is called a *generation*. It is denoted $\mathcal{G}_i = (\mathcal{N}_i, \mathcal{L}_i)$ where the index i represents the sequence number of the iteration. Output of an iteration is a new graph of stable PSM states and transitions build upon the input generation. The new generation passes to the next iteration as input. Input of the first iteration is a set of initial stable PSM transitions generated by the first phase together with all contained stable PSM states formed into the *initial generation* $\mathcal{G}_0$. i-th iteration contains the following steps.

- Define the next generation.

```
$\mathcal{N}_{i+1}$ := $\emptyset$
$\mathcal{L}_{i+1}$ := $\emptyset$
$\mathcal{G}_{i+1}$ := $(\mathcal{N}_{i+1}, \mathcal{L}_{i+1})$
```

- Pass all closed stable PSM states which are not compatible with any other stable PSM state and its stable PSM transitions to the new generation.

```
foreach $A \in \mathcal{N}_i$ : $closed(A) \;\wedge\; (\nexists C \in \mathcal{N}_i : A \sim C)$
    $\mathcal{N}_{i+1}$ := $\mathcal{N}_{i+1} \cup \{A\}$
    foreach $(A, B) \in \mathcal{L}_i$
        $\mathcal{N}_{i+1}$ := $\mathcal{N}_{i+1} \cup \{B\}$
        $\mathcal{L}_{i+1}$ := $\mathcal{L}_{i+1} \cup \{(A, B)\}$
```

- Choose all compatible pairs of stable PSM states and add their combination to the next generation.

```
foreach $A \in \mathcal{N}_i, B \in \mathcal{N}_i$ : $A \sim B$
    $\mathcal{N}_{i+1}$ := $\mathcal{N}_{i+1} \cup \{A \circledast B\}$
```

- For the stable PSM states A and B from the previous step choose all stable PSM transitions with A or B as the source stable PSM state and add a combination with the other stable PSM state to the next generation.

```
foreach $C \in \mathcal{N}_i$ : $(A, C) \in \mathcal{L}_i$
    $(D, E)$ := $B \circledast (A, C)$
    $\mathcal{N}_{i+1}$ := $\mathcal{N}_{i+1} \cup \{D, E\}$
    $\mathcal{L}_{i+1}$ := $\mathcal{L}_{i+1} \cup \{(D, E)\}$
foreach $C \in \mathcal{N}_i$ : $(B, C) \in \mathcal{L}_i$
    $(D, E)$ := $A \circledast (B, C)$
    $\mathcal{N}_{i+1}$ := $\mathcal{N}_{i+1} \cup \{D, E\}$
    $\mathcal{L}_{i+1}$ := $\mathcal{L}_{i+1} \cup \{(D, E)\}$
```

The algorithm ends when there is no compatible pair of stable PSM states left in the generation $\mathcal{G}_i$. This *last generation* is output of the algorithm.

6.2.8 Finiteness

Lemma 6.31. *There exists a generation with each stable PSM state closed.*

Proof. Let us focus on the number of non-λ values in a stable PSM state and denote the number as $\nu(A)$ where A is the analyzed stable PSM state. The situation when a stable PSM state is created in a new generation can be of two kinds.

- The stable PSM state D is created as a combination of compatible stable PSM states A and B. A situation $(A \succ B \ \vee \ B \succ A) \ \wedge \ (\nu(A) = \nu(B))$ is not possible because it means that $A = B$ and states in a generation are unique.
  - If $A \succ B$ then $\nu(D) = \nu(A) > \nu(B)$.
  - If $B \succ A$ then $\nu(D) = \nu(B) > \nu(A)$.
  - Else $\nu(D) \in [max\{\nu(A), \nu(B)\} + 1; \nu(A) + \nu(B) - 1]$
- The stable PSM state D is created as target of the combination of a stable PSM state A and a stable PSM transition (C, B). A situation $\forall i : (\beta_i = \lambda \Leftrightarrow \alpha_i = \lambda) \ \wedge \ (\nu(C) = \nu(B))$ is not possible because it means that $A = C$ and states in a generation are unique.
  - If $\forall i : (\beta_i \neq \lambda \Rightarrow \alpha_i \neq \lambda)$ then $\nu(D) = \nu(A) > \nu(C)$.
  - If $\forall i : (\alpha_i \neq \lambda \Rightarrow \beta_i \neq \lambda)$ then $\nu(D) = \nu(B) > \nu(C)$.
  - Else $\nu(D) \in [max\{\nu(A), \nu(B)\} + 1; \nu(A) + \nu(B) - 1]$.

Let us denote a minimal ν in the i-th iteration as $\nu_i^{min} = min\{\nu(A) \mid A \in \mathcal{N}_i\}$. From the analysis above, it can be deduced that

$$\nu_i^{min} < \nu_{i+1}^{min}$$

In the worst case scenario, when $\nu_0^{min} = 1$, the minimal ν in the $(n-1)$-th generation (n is the number of properties) is

$$\nu_{n-1}^{min} = n$$

But it means that each stable PSM state in this generation is closed. □

Corollary 6.32. *The algorithm is finite.*

Proof. According to 6.31 there exists a generation with all stable PSM states closed. There is not any compatible pair of stable PSM states in this generation. □

6.2.9 Correctness

The goal of this section is to prove that output of the algorithm is well defined stable property state machine. The last generation is the output. First, it is shown that the last generation is equivalent to a stable property state machine. Than it is proved that the state machine is well defined.

Lemma 6.33. *The last generation contains only stable PSM states and stable PSM transitions.*

Proof. The initial generation contains only stable PSM states and initial stable PSM transitions. Combinations of stable PSM states and stable PSM transitions produce stable PSM states and stable PSM transitions again. By the principle of induction there are only stable PSM states and stable PSM transitions in the last generation. □

Lemma 6.34. *Each stable PSM state in the last generation is closed.*

Proof. It ensues directly from the algorithm finiteness (6.32). □

Corollary 6.35. *The last generation is equivalent to a stable property state machine.*

Proof. The previous lemmata and theorems 6.33, 6.34, 6.17 and 6.21 directly imply that the definition 6.8 is met. □

It remains to prove that each stable PSM state in the last generation has all outbound transitions defined. For this issue, the stable PSM state definition has to be extended. The purpose of the extension is auxiliary.

Definition 6.36. An *extended PSM state* $\overrightarrow{A}$ is a tuple

$$\overrightarrow{A} = (A, o(A))$$

where A is a stable PSM state and $o(A)$ is a vector of values $o(\alpha)$ from the domain $\{\lambda, 1\}$ with the same cardinality as A. The vector $o(A)$ is called an *output vector* of the extended PSM state. An extended PSM state can be denoted as follows.

$$\overrightarrow{A} = \left(\begin{pmatrix} \alpha_1 \\ \dots \\ \alpha_i \\ \dots \end{pmatrix}, \begin{pmatrix} o(\alpha_1) = \lambda \\ \dots \\ o(\alpha_i) = 1 \\ \dots \end{pmatrix} \right) = \begin{pmatrix} \alpha_1 \\ \dots \\ \alpha_i \triangleright \\ \dots \end{pmatrix}$$

Operations and relations defined on stable PSM states are basically the same on extended PSM states.

- $\overrightarrow{A} \sim \overrightarrow{B} \iff A \sim B \ \wedge \ o(a) \sim o(B)$
- $\overrightarrow{A} \succ \overrightarrow{B} \iff A \succ B \ \wedge \ o(a) \succ o(B)$
- $\overrightarrow{A} \circledast \overrightarrow{B} = (A \circledast B, o(a) \circledast o(B))$

- $\overrightarrow{C} \circledast (\overrightarrow{A}, \overrightarrow{B}) = (\overrightarrow{C} \circledast \overrightarrow{A}, (B', o(B)))$ where B' is a stable PSM state constructed in 6.29.

Theorem 6.37. *Arbitrary output vectors of the same cardinality are always compatible.*

Proof. The domain of variables contained in the vectors is $\{\lambda, 1\}$ and these values can be combined with each other without limitation. □

Definition 6.38. Assume a property set $\mathcal{P} = \{p_1, \dots p_n\}$, a set of all direct properties $\mathcal{P}^{Di} \subseteq \mathcal{P}$ and a generation $\mathcal{G}_k = (\mathcal{N}_k, \mathcal{L}_k)$. An arbitrary stable PSM state $A = (\alpha_1, \dots, \alpha_n)$ defined on $\mathcal{P}$ contained in the generation $\mathcal{G}_k$ can be extended to a PSM state $\overrightarrow{A}$ by the following rule.

$$\forall i = 1, \dots, n$$

$$o(\alpha_i) = \begin{cases} 1 & p_i \in \mathcal{P}^{Di} \wedge \\ & \forall \gamma \in \mathcal{D}(p_i) : \exists (A, B) \in \mathcal{L}_k : characteristic((p_i, \gamma), (A, B)) \\ \lambda & else \end{cases}$$

What does it mean? In fact, output of each direct property is marked with the symbol $\triangleright$ if and only if all possible outbound transitions with the property acting like the transition characteristic are defined in the generation.

$$\begin{pmatrix} 0 \triangleright \\ \lambda \\ \lambda \end{pmatrix} \longrightarrow \begin{pmatrix} 1 \\ \lambda \\ 1 \end{pmatrix}$$

Let us define a simulation of the algorithm on extended PSM states with the results equivalent to the results of the original algorithm.

Definition 6.39. *i-th algorithm iteration on extended PSM states* is defined by the following steps.

- Run the i-th iteration of the original algorithm with extended PSM states on input. Use operations and relations defined in 6.36.
- Choose an arbitrary pair of extended PSM states $\overrightarrow{A}, \overrightarrow{B} \in \mathcal{N}_{i+1}$ which fulfills the condition $A = B$ and combine them into one as follows.

  $$(\ A,\ o(A) \circledast o(B)\ )$$

  Add the result to the $(i+1)$-th generation with the input and output stable PSM transitions inherited from the extended PSM states $\overrightarrow{A}$ and $\overrightarrow{B}$. Finally, remove the extended PSM states $\overrightarrow{A}$ and $\overrightarrow{B}$ from the $(i+1)$-th generation.
- Iterate the previous step until there is a pair of extended PSM states which fulfills the condition of the previous step.

Lemma 6.40. *Let us simulate one iteration of the algorithm on a generation composed of extended PSM states according to 6.38. The new generation is composed of extended PSM states which meet the definition 6.38 as well.*

Proof. Assume a situation when two extended PSM states $\overrightarrow{A}$ and $\overrightarrow{B}$ are combined creating an extended PSM state $\overrightarrow{C}$ in the new generation. Without loss of generality assume that $\overrightarrow{A}$ has an output mark on the index i. It means, that all possible outbound transitions are defined for the property p_i (6.38). This output mark is inherited to $\overrightarrow{C}$. But the algorithm combines B with all stable PSM transitions having A as a source and thus passes the corresponding transitions having C as a source to the new generation. □

The presented extension is a convenient instrument to prove that the property state machine is well defined. The idea is shown in the following lemma.

Lemma 6.41. *Assume a property set $\mathcal{P} = \{p_1, \dots p_n\}$, a set of all direct properties $\mathcal{P}^{Di} \subseteq \mathcal{P}$ and the last generation $\mathcal{G}_k$ with stable PSM states extended according to 6.36 and 6.38. If an arbitrary extended PSM state $\overrightarrow{A} = (A, o(A))$, $A = (\alpha_1, \dots, \alpha_n)$ in the generation $\mathcal{G}_k$ holds*

$$\forall i, p_i \in \mathcal{P}^{Di} : \; o(\alpha_i) = 1$$

then the last generation $\mathcal{G}_k$ is well defined.

Proof. It is implied directly from the definitions 6.9 and 6.38. □

Another auxiliary lemma makes possible to decompose a stable PSM state existing in the last generation to (initial) stable PSM states from the initial generation.

Lemma 6.42. *Assume a closed stable PSM state $A = (\alpha_1, \dots, \alpha_n)$ and an arbitrary but fixed index i. There exists a stable PSM state $B = (\beta_1, \dots, \beta_n)$ in the initial generation $\mathcal{G}_0$ compatible with A which holds $\alpha_i = \beta_i \neq \lambda$.*

Proof. Assume a property set $\mathcal{P} = \{p_1, \dots p_n\}$ and the initial generation $\mathcal{G}_0 = (\mathcal{N}_0, \mathcal{L}_0)$.

- The property p_i is direct. The initial generation $\mathcal{G}_0$ contains all possible initial stable PSM transitions, thus there exists a set of initial stable PSM transitions $\{(B_1, C), \dots, (B_m, C)\}$ with the same characteristics (p_i, α_i). All the transitions have the same target but different sources. Each source holds the condition described in 6.22. All sources together represent all stable value permutations of properties in $Re(p_i) \cup \{p_i\}$. Because of stability of A there exists k that B_k matches the theorem.

- The property p_i is indirect. Because of stability of A the value α_i is transitively k-supported where k is the index of a direct property. For the value α_k, there exists an initial stable PSM transition $(B, C) \in \mathcal{L}_0$ with the characteristic (p_k, α_k). The stable PSM state C matches the theorem.

$\square$

Finally, let us prove that the last generation is well defined. Assume a closed stable PSM state A from the last generation. Let us build a set of initial stable PSM states $\mathcal{I}$ which are compatible with A by the following algorithm.

```
$\mathcal{I}$ := $\emptyset$
foreach $i = 1, \dots, n$
   foreach $B \in \mathcal{L}_0$:  found according to 6.42
      $\mathcal{I}$ := $\mathcal{I} \cup \{B\}$
```

Assume $\mathcal{I} = \{B_1, B_2, \dots, B_m\}$. The stable PSM states in $\mathcal{I}$ are pairwise compatible (combination creates a stable PSM state more similar to A) and combined with each other they create the stable PSM state A.

$$A = (B_1 \circledast (B_2 \circledast (B_3 \dots (B_{m-1} \circledast B_m) \dots)))$$

Let us extend the stable PSM states $B_k \in \mathcal{I}$ and the PSM state A according to 6.38. The extended PSM states in $\mathcal{I}$ contains together all possible output marks. If the formula "A inherits all output marks from the initial generation"

$$o(A) = (o(B_1) \circledast (o(B_2) \circledast (o(B_3) \dots (o(B_{m-1}) \circledast o(B_m)) \dots))) \qquad (*)$$

is valid than the last generation is according to 6.41 well defined and the algorithm is correct.

Let us prove the $(*)$ formula validity. For contradiction, assume that $\exists k : o(A) \not\succ o(B_k)$. It means that $\exists i : o(\alpha_i) = \lambda \ \wedge \ o(\beta_i^k) = 1$ in the extended PSM states $\overrightarrow{A} = (A, o(A))$, $A = (\alpha_1, \dots, \alpha_n)$ and $\overrightarrow{B_k} = (B_k, o(B_k))$, $B_k = (\beta_1^k, \dots, \beta_n^k)$.

Let us simulate one iteration of the algorithm on the extended initial generation and aim at the set $\mathcal{I}$. A new generation of extended PSM states is created. All extended PSM states in the new generation which are created as a combination of two extended PSM states from $\mathcal{I}$ are pairwise compatible again. The extended PSM state $\overrightarrow{B_k}$ is compatible with an arbitrary other extended PSM state $\overrightarrow{B_l} \in \mathcal{I}$ so there exists a combination $\overrightarrow{C} = \overrightarrow{B_k} \circledast \overrightarrow{B_l}$, $\overrightarrow{C} = (C, o(C))$, $C = (\gamma_1, \dots, \gamma_n)$ in the new generation. This combination inherits the output mark from $\overrightarrow{B_k}$, it means that $o(\gamma_i) = 1$.

By the principle of induction the algorithm can be simulated until the generation which contains the extended PSM state $\overrightarrow{A}$ is reached. There exists an extended PSM state $\overrightarrow{C}$ in this generation which inherits the output mark from B_k and it is not $\overrightarrow{A}$. But $\overrightarrow{A} \sim \overrightarrow{C}$. It is in contradiction with the presumption that the reached generation is the last generation.

$\blacksquare$

Corollary 6.43. *The algorithm is correct.*

6.2.10 Complexity

Theorem 6.44. *Number of stable PSM states in the last generation is polynomial to the number of stable PSM states in the initial generation.*

Proof. Assume a generation $\mathcal{G}_i$ with m stable PSM states and its succeeding generation $\mathcal{G}_{i+1}$. Let us evaluate an upper bound of stable PSM states count in the generation $\mathcal{G}_{i+1}$. Assume two compatible stable PSM states A and B in the generation $\mathcal{G}_i$ and combine them as the algorithm defines creating a stable PSM state $A \circledast B$ in the generation $\mathcal{G}_{i+1}$. A is the source of several transitions which can direct to at most m unique targets. All the transitions are combined with B so their targets ("combined" with B according to 6.29) are included into the generation $\mathcal{G}_{i+1}$. The same idea can be applied symmetrically to transitions with B as the source. It means that for a pair of compatible stable PSM states in the generation $\mathcal{G}_i$, there are at most

$$2 \cdot m + 1$$

stable PSM states in the generation $\mathcal{G}_{i+1}$. There are at most

$$\binom{m}{2}$$

pairs of compatible stable PSM states in the generation $\mathcal{G}_i$. Each stable PSM state can be also passed to the generation $\mathcal{G}_{i+1}$ if it is not compatible with any other stable PSM state in the generation $\mathcal{G}_i$. It means that an upper bound can be evaluated as

$$(2m+1) \cdot \binom{m}{2} + m \;=\; m^3 - \frac{1}{2}m^2 + \frac{1}{2}m \;\leq\; m^3 \;(m \in \mathbb{N})$$

Assume an application with n properties. This number is now a constant. Then assume that the initial generation $\mathcal{G}_0$ contains M stable PSM states. According to 6.31, the algorithm makes at most $n-1$ main iterations. It means that number of stable PSM states in the generation $\mathcal{G}_{n-1}$ is at most

$$M^{(3^{n-1})}$$

□

Corollary 6.45. *Time and space complexity of the algorithm's second phase is polynomial with respect to size of the initial generation.*

6.3 Hierarchical Mode Automaton

The property state machine constructed as described is able to react to events and properly adjust the property values. However, the question which components and connections are enabled and which values of real-time attributes are applied in a particular property state is not answered yet.

Assume a components composed into a hierarchy with component modes as defined in Chapter 5. Let us focus just on component modes now. The modes create a hierarchical automaton. It means that a mode can contain other modes formed into a smaller automaton. A mode can contain even more than one nested automaton.

There is resemblance with hierarchical automata defined in UML [17]. Let us use UML terminology in the following text and define a subset of the automata suitable for the purpose of this chapter.

Definition 6.46. A *HMA region* is a triplet $(\mathcal{M}, M_0, \tau)$ where

- $\mathcal{M}$ is a nonempty set of HMA modes contained in the region. The HMA mode is defined below.
- $M_0 \in \mathcal{M}$ is an *initial HMA mode.*
- $\tau : \mathcal{M} \times \mathcal{C} \mapsto \mathcal{M}$ is a *HMA transition function.* $\mathcal{C}$ is a set of mode conditions of all modes in $\mathcal{M}$.
$$\mathcal{C} = \{\Gamma_{M_i} \mid \forall M_i \in \mathcal{M}\}$$
The transition function defines which target HMA mode is active after a mode condition is met in an active HMA mode.

A *HMA mode M* is a pair $(N, \mathcal{R})$ where

- N is a component mode.
- $\mathcal{R}$ is a set of HMA regions. This set is denoted $M(\mathcal{R})$.

The HMA mode M is *primitive* if and only if $M(\mathcal{R})$ is empty. Otherwise it is *composite. M* is *orthogonal* if and only if $M(\mathcal{R})$ has two or more elements.

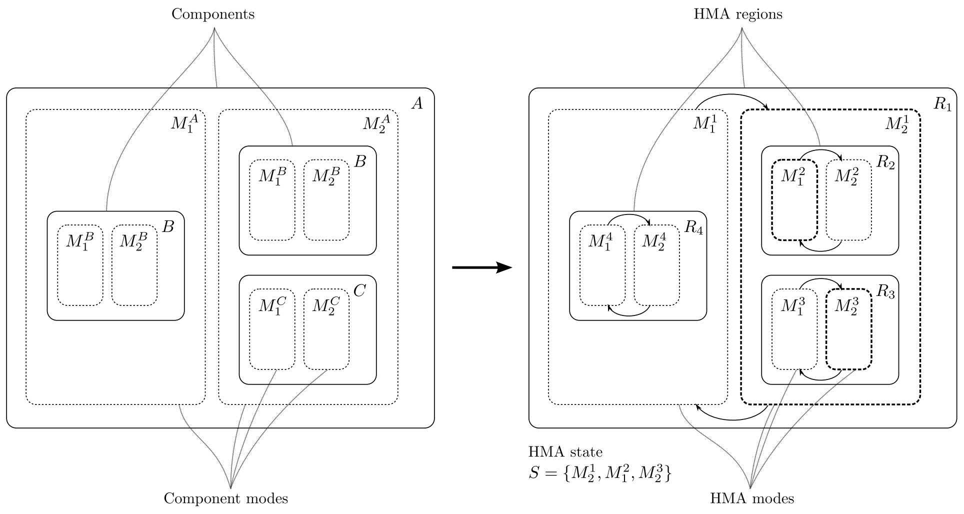


Figure 6.3: Hierarchical mode automaton

A region can be understood as a representation of one composite component. It also encapsulates one finite automaton defined at a single level of the hierarchy. A HMA mode is an extension of a simple component mode. The extension adds a possibility to enclose some other regions, thus creates the mode automata nesting. Figure 6.3 shows an example of a component hierarchy with a corresponding hierarchical mode automaton.

Basic child/parent relations between modes and regions can be observed on the hierarchical structure of an automata.

Definition 6.47. Assume two sets of HMA regions

$$\mathcal{R}_1 = \{R_{1,1}, R_{1,2}, \dots\},\ \mathcal{R}_2 = \{R_{2,1}, R_{2,2}, \dots\}$$

and a HMA mode $M \in R_{1,i} \in \mathcal{R}_1$ for an arbitrary but fixed index i. Assume that $M = (N, \mathcal{R}_2)$ as well. For an arbitrary but fixed index j.

- the HMA region $R_{1,i}$ is called a *parent region* of the region $R_{2,j}$.
- the HMA region $R_{2,j}$ is called a *child region* of the region $R_{1,i}$.
- the HMA mode M is called a *parent mode* of the region $R_{2,j}$.
- the HMA region $R_{2,j}$ is called a *child region* of the mode M.

Definition 6.48. A *hierarchical mode automaton* is one HMA region called a *top-level region.*

Like every other automaton a hierarchical mode automaton defines a state in which it can be at runtime. The state of the hierarchical mode automaton is not just one active mode but a collection of modes which contains active modes for all transitively enabled subcomponents. Each state of the hierarchical mode automaton represents one configuration of the application. The state can be defined by the following recurrent definition.

Definition 6.49. Assume a hierarchical mode automaton H. A set of HMA modes $S = \{M_1, M_2, \dots\}$ is called a *HMA state* if and only if

- there exists just one mode in the state S from the top-level region.

$$\exists!\ M_j \in S :\ M_j \in H(\mathcal{M})$$

- there exists just one mode in the state S from each region whose parent mode is in the state S.

$$\forall M_k \in S, M_k = (N, \mathcal{R}_k) :\ (\forall R_i \in \mathcal{R}_k :\ (\exists!\ M_j \in S :\ M_j \in R_i(\mathcal{M})\ ))$$

It can be said that the mode M_j *represents* the region R_i (H) or that the region is *represented* by the mode in the state S.

As described in Section 5.2.3, a link between component modes and properties is established by a special mode property that exists in a well defined property set of a multi-moded component. The HMA state corresponding to a property state can be found using the special mode properties as described by the following definition.

Definition 6.50. Assume a property set $\mathcal{P} = \{p_1, \dots, p_n\}$, a property state $A = (\alpha_1, \dots, \alpha_n)$ defined on $\mathcal{P}$ and a HMA state $S = \{M_1, \dots, M_m\}$. The HMA state S is *corresponding* to the property state A if and only if

$$\forall M_i \in S :\ \alpha_j = M_i$$

where α_j is a value of a property p_j in the property state A and p_j is a property corresponding to the mode M_i according to 5.12.

6.4 HMA Algorithm

An algorithm which finds a HMA state corresponding to a property state is proposed in this section. The first part presents the algorithm itself, the second part proves that the algorithm ends in finite time with correct result and shows its time and space complexity.

6.4.1 Algorithm

Input of the algorithm is a property state A and a hierarchical mode automaton H. Output of the algorithm is a HMA state S corresponding to the property state A.

The algorithm executes the following steps. First, initiate a set of HMA regions $\mathcal{R}$ and a set of HMA states S.

```
$\mathcal{R}$ := $\{H\}$
$S$ := $\emptyset$
```

Then iterate the following steps until the set $\mathcal{R}$ is empty. The set S is output of the algorithm.

```
Let $R_i$ be an arbitrary but fixed region in $\mathcal{R}$
$\mathcal{R}$ := $\mathcal{R} \smallsetminus \{R_i\}$
foreach $M_j \in R_i$
    find corresponding property $p_k$ and its value $\alpha_k \in A$
    if $\alpha_k = M_j$ then
        $S$ := $S \cup M_j$
        $\mathcal{R}$ := $\mathcal{R} \cup M_j(\mathcal{R})$
```

6.4.2 Finiteness, Correctness and Complexity

Theorem 6.51. *Let a finite hierarchical mode automaton H and a property state A be input of the algorithm. Then the algorithm is finite.*

Proof. The modes contained in H form together a finite tree. The algorithm traverses the tree and each node is inspected at most once. □

Theorem 6.52. *Let a finite hierarchical mode automaton H and a property state A be input of the algorithm and let the algorithm end with the set S on output. Then S is a HMA state corresponding to A.*

Proof. As soon as some region R is inspected during the algorithm process, there exists a mode M in the resulting set S which represents the region R.

Let us show that S is a HMA state. Because of the initial content of the set $\mathcal{R}$ the top-level region is surely inspected and represented in the result. It fulfills the first requirement to the definition 6.49. For each mode added into the result the algorithm adds all its child regions to $\mathcal{R}$ for a future inspection. So these regions are represented in the result as well. It fulfills the second requirement to the definition 6.49.

It remains to prove that the HMA state S is corresponding to the property state A. A corresponding property of each mode M in S is evaluated with the values from A and the mode M is added to S if and only if the evaluation fulfills the definition 6.50. $\square$

Theorem 6.53. *Let a finite hierarchical mode automaton H and a property state A be input of the algorithm. Time complexity of the algorithm is linear to the number of elements transitively contained in H.*

Proof. Each HMA region and HMA mode transitively contained in H is visited by the algorithm as most once. $\square$

6.4.3 Enable and Attribute Lists Association

Each component mode defines a subcomponent and connection enable list and subcomponent attribute list, i.e. sets of components and connections which are enabled and values of real-time attributes which are valid when the mode is active. A HMA state represents a set of active component modes. It means that the set of components enabled in the HMA state is simply an union of the subcomponent enable lists or of all modes contained in the state. The same idea can be applied to connection enable lists and component attribute lists.

Definition 6.54. Assume a HMA state $S = (M_1, \ldots, M_m)$.

- A set of components $M_1(\mathcal{S}^e) \cup \cdots \cup M_n(\mathcal{S}^e)$ is called a *HMA state component enable list.*
- A set of components $M_1(C^e) \cup \cdots \cup M_n(C^e)$ is called a *HMA state connection enable list.*
- A set of value definitions $M_1(R) \cup \cdots \cup M_n(R)$ is called a *HMA state component attribute list.*

Finally, assume a well defined stable property state machine. For each stable property state contained in the state machine, a corresponding HMA state can be found and its component enable list, connection enable list and component attribute list define a configuration of the application valid for the property state.

Chapter 7

Runtime

This chapter provides an image how the application looks like at runtime and shows details about the mechanism runtime architecture introduced in Section 4.4.7. As the oracle based approach for the network reaction representation is mainly discussed in the thesis, the architecture variant using the oracle is assumed in the following description.

Application is at runtime composed of deployed components and connections, i.e. with their instances. A configuration of the application corresponds to a HMA state (6.49) or more precisely to its component enable list, connection enable list and attribute list (6.54). Deployed components and connections are formed either into a hierarchical component model again or to a flat component model as described in Section 3.4. Ideas presented in this chapter can be applied to both hierarchical and flat component models with minimal modifications.

7.1 Structural Requirements

In every moment of the runtime reconfiguration, *runtime structural requirements* similar to the meta-model structural requirements (5.4) have to be satisfied.

(1) An arbitrary require interface of an arbitrary enabled component has to be connected to exactly one compatible provide interface of another enabled component (for both hierarchical and flat component models) or to exactly one require interface of a parent component (for a hierarchical component model only).

(2) An arbitrary provide interface of an arbitrary enabled composite component has to be connected to exactly one provide interface of an enabled subcomponent (for a hierarchical component model only).

(3) An arbitrary enabled connection has to connect only interfaces defined on enabled components (for both hierarchical and flat component models).

Two observations about the process of enabling and disabling application parts can be made as corollaries implied from the structural requirements.

First, in the time when a component is disabled, also all connections connected to its provide interfaces are disabled, thus components requiring a service of the disabled component are either rerouted to another components or disabled as well.

Second, in the time when a component is enabled, each of its require interfaces is connected to a component providing the required service. This component is either enabled before the requiring component or it is enabled with the requiring component at the same time.

7.2 Architecture

As the runtime architecture is divided into the three parts – gathering the values, the network reaction representation and the reconfiguration based on a mode change protocol – there exist three components, each of them realizing one of the activities. By encapsulating the activity to a single component, the flexibility of the architecture is preserved. The component defines an imutable interface but its implementation can be switched according to the situation, i.e. the type of the network reaction representation and the mode change protocol can be easily replaced. The whole architecture is shown in Figure 7.1.

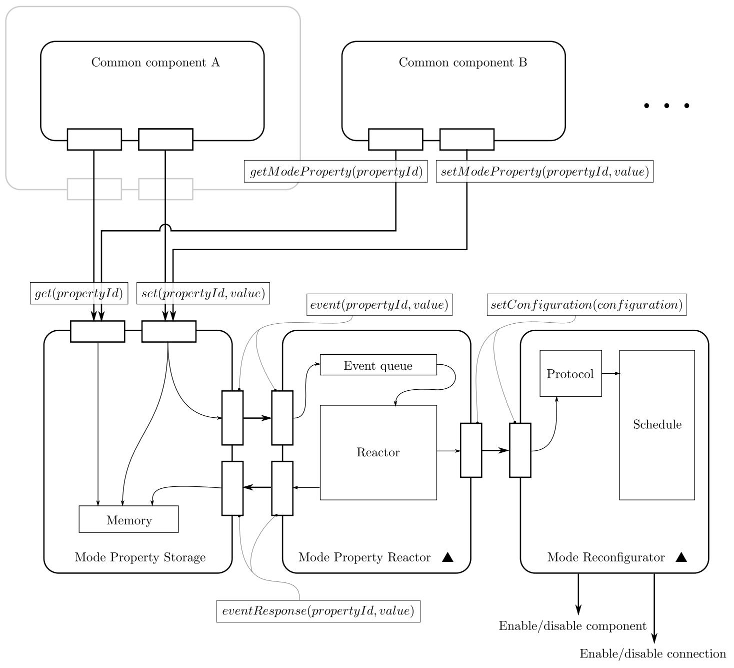


Figure 7.1: Architecture of the mode change mechanism at runtime

The first part of the architecture is represented by the component `Mode Property Storage`. The component acts like memory that stores current property values. A connection to common components is realized through the interfaces `set` and `get`. Compatible interfaces `setModeProperty` and `getModeProperty` are automatically generated for each common component at deploy phase. A common component is able to send a new value through the `setModeProperty` interface and gain the current value through the `getModeProperty` interface. When a new value is sent it is stored in memory and an event is triggered. The event enters the second architecture part as input. A common component cannot set values of indirect properties, hence the component `Mode Property Storage` can store and handle just direct properties.

The second part is represented by the component `Mode Property Reactor`. On the basis of an input event a network reaction is simulated and a stable property state is found. Output of the process is an update of property values and a new configuration of the application. Current property values are updated in the first architecture part and the new configuration is transfered to the third architecture part as input. As proposed in Chapter 4 this architecture part incorporates own extra task. For this reason, an event queue is present in order to synchronize data flow between the extra task and common component's tasks.

The component `Mode Reconfigurator` represents the third architecture part. When a new configuration is on input, it is compared to the current configuration of the application and input for a mode change protocol is created based on this. The mode change protocol finds and saves a schedule. This architecture part incorporates another extra task which just reads the schedule and enables or disables application parts at the right time.

7.3 Component Interface

It is possible that the source code of a common component uses directly the interfaces `setModeProperty` and `getModeProperty` for the property handling . However, this approach has several drawbacks. First of all, since the interfaces are generated in deploy phase they do not exist in design phase. The next problem is the parameter of the interface calling – the property identifier. The identifier has to be unique among all application properties so it has to be an automatically generated number. It means that the identifier is not human-readable. Finally, if the mechanism architecture at runtime were implemented differently the compatibility of old components could be broken.

The drawbacks can be neglected or overridden but they are still retarding for the user. A promising solution is an extension of the component API with new methods regarding to the property handling. Implementation of the methods uses the generated interfaces and is automatically generated with them in deploy phase. It covers the automatically generated property identifiers and brings opportunity to define extra functions.

The component API is extended by the following basic methods.

- `setModeProperty(propertyLocalId, value)` – Set a new value to a property.
- `getModeProperty(propertyLocalId)` – Get a value of a property.

The following extra methods are included as well.

- `setChangedModeProperty(propertyLocalId, value)` – Set a new value to a property but only if the condition that a new value is different from the current one is met. The difference to the method `setModeProperty` is that the calling of the method with the current value does not trigger an event which could cause a reconfiguration.
- `resetModeProperty(propertyLocalId)` – The current value of a property is not changed, just an event is triggered.

7.4 Event Queue

As described in Section 2.3, a real-time application cannot rely on availability of a suitable dynamic memory allocator. It is convenient to implement the event queue using static memory allocation. It means that the maximum size of the queue has to be set in advance. It can be done in this particular situation but one presumption has to be introduced.

Assume that each property can be modified by one instance of a task at most once. It means that the number of events offered to the queue by the instance is equal to the number of direct properties.

The following analysis is done for the RM scheduling algorithm. For a different algorithm the results can be different. In the worst case scenario, the number of instances of common component's tasks scheduled between two instances of the reactor task can be above bounded by the number

$$\sum_{i \in Co} \left\lceil \frac{2 \cdot T_r}{T_i} \right\rceil$$

where Co is a set of indices of all common tasks, T_i is a period of a task with index i and T_r is a period of the reactor task. With another presumption that the reactor task has higher priority than an arbitrary common task, the upper bound can be estimated to $2\,|Co|$. In this case, the maximum size of the queue is

$$2 \cdot |Co| \cdot \left|\mathcal{P}^{Di}\right|$$

where $\mathcal{P}^{Di}$ is a set of all direct properties.

7.5 Oracle Based Reactor

The component `Mode Property Reactor` is based on the property state machine which acts like an oracle for the network reaction representation. This structure is stored in the component. Besides this, the component keeps an information about current state of the machine.

Memory complexity of the property state machine can be reduced by omitting values of indirect properties in each state of the property state machine. It can be done because of the fact that the indirect properties are not stored in the first architecture part and no other architecture part uses them. However the structure of the property state machine – a number of states and transitions – has to remain unchanged.

7.6 Mode Change Protocol

The Maximum-period offset protocol can be easily used as a mode change protocol. This protocol is simple to be implemented and supports periodicity. Assuming that the meta-model structural requirements (5.4) are satisfied in the design phase, the runtime structural requirements described are also satisfied because of the fact that a reconfiguration is done at the same time (with a single offset). However, a promptness of a reconfiguration is poor under this protocol.

In order to improve the promptness, more sophisticated protocols should be incorporated. But this is out of scope of the thesis. There is an opportunity for further research. The designed architecture is at least flexible enough in order to make the other protocol's implementation easier.

7.7 Enabling and Disabling Application Parts

A real-time application cannot rely on availability of a dynamic memory allocator. It is convenient to design the reconfiguration without the dynamic memory allocation. All components enabled in an arbitrary configuration exists in memory all the time the application is executed. Disabling and enabling components is accomplished by the following methods.

- Implicitly. If the component is supposed to be disabled, there are no enabled connections to it directing transitively from an arbitrary enabled active component, thus a thread of execution cannot reach the disabled component. If the component is supposed to be enabled, all components requiring its service are connected to it. This method is valid because of the validity of the runtime structural requirements.

- Explicitly. Active components are enabled by starting the component's task and disabled by stopping the component's task.

Enabling and disabling application parts is realized using a schedule. It is a structure which maps time points to atomic operations. The component `Mode Reconfigurator` executes the atomic operations at the right time as described in the schedule by calling system API. The atomic operations can be:

- `startTask(componentTask, attributes)` – Start an active component's task with the specified real-time attributes.

- `stopTask(componentTask)` – Stop an active component's task.

- `createConnection(sourceItf, targetItf)` – Enable connection between two interfaces.
- `destroyConnection(sourceItf, targetItf)` – Disable connection between two interfaces.

Chapter 8

Implementation in SOFA HI

A proof-of-concept implementation is included in the thesis. It is based on SOFA 2 Component system or more precisely on its real-time profile SOFA HI (proposed in [1]). This chapter describes specific properties of SOFA HI and a way how to implement a support for operating modes in it.

8.1 SOFA HI Specific Characteristics

SOFA HI has several specific characteristics that differentiate it from a generic hierarchical component system described in Chapter 3. Some of the following characteristics cause that the implemented mode change concept has to be adjusted. Other characteristics are just informative and they are described in order to make the implementation intelligible.

A component's structure is divided in two entities – a frame and an architecture. A frame defines an outer interface of the component, i.e. a way how other components communicate with the component. An architecture, on the other hand, represents the component's implementation. The component can even use more than one frame but just one architecture. The component's structure division brings a question which new entities – component modes, mode properties and mode property functions – are defined in the context of a frame and which ones in the context of an architecture.

SOFA HI uses a flat component meta-model at runtime. An original hierarchical meta-model is in SOFA HI called a repository meta-model. An instance of the repository meta-model is in deploy phase flattened and transformed into an instance of an inter meta-model. A runtime architecture of the application is based on this flat inter model. Hence the mode change mechanism have to work at runtime with common components composed at one level only.

SOFA HI defines two types of components – active and passive. An active component carries its own thread of execution, defines real-time attributes and implements an infinite loop in order to create a periodical task. A passive component acts just like a service for other components. Its code is executed in the context of an active component.

Aside an application structure, a deployment plan is defined. A deployment plan is a supporting configuration which describes how the application is deployed and which values of component's configuration parameters are applied. This is the place where values of real-time attributes are defined.

8.2 Repository Meta-model Extension

A question where new entities are defined – either on a frame or on an architecture – is presented in the previous section. This section answers the question and presents an extension of the SOFA HI meta-model overall.

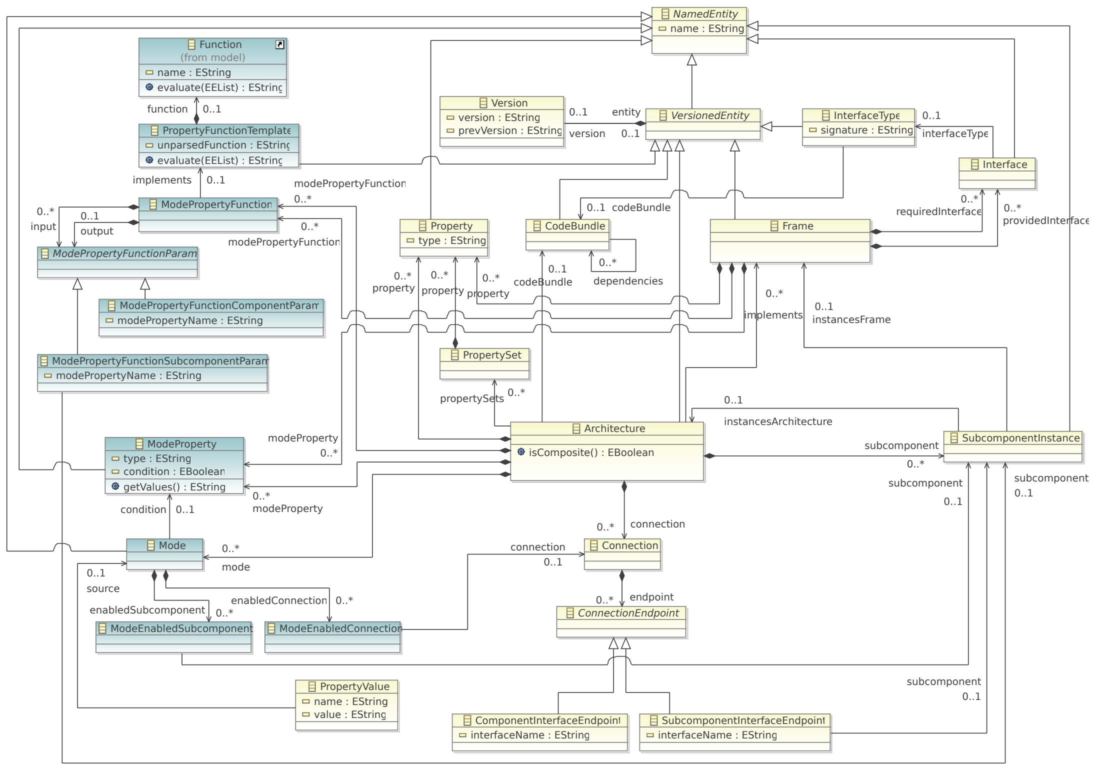


Figure 8.1: Extension of repository meta-model

Since a component mode is in fact an internal configuration of a component, it is defined on an architecture. This refers to the subcomponent and connection enable lists. Values of real-time attributes are defined in the context of deployment plan. Hence the deployment plan is the place where different values for different component modes are specified.

Mode properties are responsible for a communication with component's outer surrounding. Therefore, it is necessary to define them in the context of a frame. However, the second responsibility of mode properties refers to the mode condition. The second responsibility is in the mechanism modeled using a special mode property corresponding to the component mode. The mode property cannot be defined on a frame in order to preserve frame's independence from the component modes defined on the architecture. Since different mode properties have different responsibilities, it is convenient to define them on both entities. As a result, a set of component mode properties is an union of sets of both frame and architecture mode properties. It is up to the user to decide which properties are general enough to be defined on the frame and which are architecture-specific.

Mode property functions create relations between mode properties and thus have to be defined alike. Functions working with frame mode properties only can be defined in the context of the frame. Functions which works fully or partially with architecture mode properties have to be defined in the context of the architecture in order to preserve a concept of an independent frame. As a result, a set of component mode property functions is an union of the two sorts of functions.

Mode property function is implemented by a template of functional behavior as described in Section 4.4.2. It is possible to re-use the templates for different properties in the application or even for different applications. That is the reason why templates are stored as first-class objects in the common library. It has been discussed that templates are defined using the special PFL language. Similarly, a (semantic) meta-model of templates is not part of the repository meta-model and the repository meta-model just refers it through a definition of a mode property function.

The entire extension of the meta-model is shown in Figure 8.1. All new entities are highlighted.

8.3 ADL Extension

SOFA 2 ADL is used as an architecture description language of SOFA HI system. This section describes all extensions of SOFA 2 ADL intended to model new entities of meta-model.

An architecture is extended of component modes. Therefore an ADL element `architecture` can contain multiple occurrences of a newly defined element `mode`. A component mode has a name and defines a list of enabled subcomponents and connections. To represent this fact, an element `mode` contains an attribute `name` and multiple occurrences of subelements `enabled-sub-comp` representing an enabled subcomponent and `enabled-connection` representing an enabled connection. To make the referencing of component's connections in ADL possible, it is necessary to add an optional attribute `name` to the element `connection`.

```
<architecture name=‘‘architecture-name’’ frame=‘‘frame-ref’’ ...>
  <sub-comp name=‘‘subcomponent-name’’ frame=‘‘frame-ref’’ .../>
  <connection name=‘‘connection-name’’ >
    <endpoint sub-comp=‘‘subcomponent-name’’ itf=‘‘interface-name’’ />
  </connection>
  <mode name=‘‘mode-name’’>
    <enabled-sub-comp sub-comp=‘‘subcomponent-name’’ />
```

```
      <enabled-connection connection=‘‘connection-name’’ />
   </mode>
   ...
</architecture>
```

The second part of a component mode definition is in a deployment plan, where differences of real-time attribute values are defined. An ADL element `depl-subcomponent` can contain multiple occurrences of a newly defined element `depl-mode`. This element contains an attribute `name` referencing a particular component mode. It also contains multiple occurrences of element `depl-prop-value`. Values defined in the context of an element `depl-mode` are applied in the referenced component mode.

```
...
<depl-subcomponent name=‘‘subcomponent-name’’ ...>
   <depl-mode name=‘‘mode-name’’>
      <depl-prop-value name=‘‘prop-name’’>value</depl-prop-value>
   </depl-mode>
   ...
</depl-subcomponent>
...
```

Mode properties and mode property functions are defined on both a frame and architecture. To model this relations, an ADL element `frame` (`architecture`) can contain multiple occurrences of an element `mode-prop` and multiple occurrences of an element `mode-prop-function`. Each element `mode-prop` represents a mode property and contains an attribute `name` and an attribute `type`. Each element `mode-prop-function` represents a mode property function and contains an attribute `impl` which bounds it with a function template. This element can contain subelements representing function's input and output properties. It contains exactly one element `output` and multiple occurrences of element `input`. Both elements can be in two variants. The first variant represents local property reference. An element contains an attribute `prop` which refers to a mode property defined in a local frame or architecture. The second variant represents an external reference. An element contains an attribute `sub-comp` referring a particular subcomponent and an attribute `prop` which links it with a mode property defined on the referenced subcomponent's frame.

```
<frame name=‘‘frame-name’’>
   <mode-prop name=‘‘mode-prop-name’’ type=‘‘mode-prop-type’’ />
   <mode-prop-function impl=‘‘pft-ref’’>
      <input prop=‘‘mode-prop-name’’ />
      <input sub-comp=‘‘sub-comp-name’’ prop=‘‘mode-prop-name’’ />
      <output prop=‘‘mode-prop-name’’ />
   </mode-prop-function>
   ...
</frame>
```

A template of function's behavior is defined as a completely new object. In ADL it is represented by an element `property-function-template` with an attribute `name`. This element contains a description of the function written in PFL.

```
<property-function-template name=‘‘property-function-name’’>
   <![CDATA[ ...  PFL code ...  ]]>
</property-function-template>
```

8.4 Deploy Phase and Runtime

Once an application is designed, a deploy phase comes. All prerequisites are fulfilled at this time to generate the runtime mechanism components `Mode Property Storage`, `Mode Property Reactor` and `Mode Reconfigurator` as proposed in Chapter 7. The components are inserted into the repository model as standard components. A property state machine necessary for generating an oracle based reactor is found by the PSM-2 algorithm examined in Chapter 6.

The process of compilation and linking do not need to be modified at all. Some modifications referring to the component and system API and implementation of low-level enabling and disabling of application parts have to be done at runtime level according to Chapter 7.

Chapter 9

Case Study

This chapter introduces an example application which is used for demonstration of the designed concept. It is a non-trivial example with component composition to the depth of three levels. There are two levels where component modes are defined and it gives opportunity to define properties connected by non-trivial property functions. On the other hand it is not too difficult in order to preserve apprehensibility.

9.1 Specification

The example is based on the existence of LEGO MINDSTORMS set and its NXT control technology [15]. This technology allows to build independent robots which are able to apperceive an environment and act in it using different sensors and actuators. Central unit of the technology is an NXT brain. Furthermore, the example assumes a color sensor which enables the robot to distinguish between colors, a touch sensor which detects when it is being pressed and two servo motors which move with the robot in the environment.

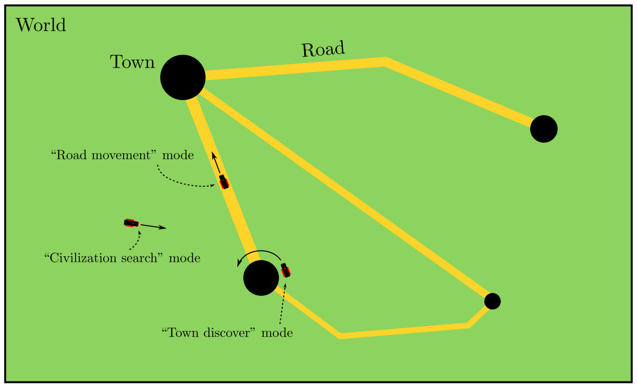


Figure 9.1: Example world

In front of the robot there is a bumper which uses the touch sensor for collision detection. The color sensor is directed downwards so the robot can "see" what color it is standing on. The two servo motors are used as engines for a left and right creeper track. It allows the robot to move and turn round.

The example application implements robot's specific control logic. It is able to exist in a world which is composed of areas with different colors and bordered with a wall. Each color has a specific interpretation. There are black circles which represent towns, yellow lines which represent roads and connect towns, and green rest of the world which can be understood as grass. When the robot is dropped on grass it searches for a civilization, i.e. a road or a town. Once it finds something, it searches the world. It follows roads and discovers towns. When the world is completely discovered the robot stops and switches to a standby mode.

9.2 Component Model

There can be observed three operating modes – a civilization search, a road movement and a town discovering. Besides this, another operating modes take part in the application. There can be a system initialization and a standby mode which is active once a world searching is done. It is reasonable to model the two sets of modes on a different component level, so it creates the following mode hierarchy.

- Initialization – The robot is booting up.
- World search – The robot discovers towns and roads and creates a world image in memory.
  - Civilization search – The first town or road is searched for.
  - Road movement – A road is followed hoping there is a town at the end.
  - Town discovering – A town is analyzed and roads directing from the town are discovered.
- Standby – The robot is done with searching and stops.

The high-level mode set is controlled using two simple boolean flags. The first one describes if an initialization is already done. The second one indicates if there is a complete world image in memory. Initialization mode is the first active. Completed initialization indicates a switch to the world search mode. When the world image in memory is considered to be complete the robot switches to the standby mode.

A control of the low-level mode set – switching between civilization search, road movement and town discovering – is not so simple. It uses an information which color is the robot standing on, an indicator if an algorithm for town discovering is already done and the active mode itself. If the robot searches for a civilization and finds a road it switches to the road movement mode, if it finds a town it switches to the town discovering mode. If it is following a road and finds a town it switches to the town discovering mode as well. Finally if it is discovering a town and this algorithm is done, it moves the robot on the next road in schedule and indicates this fact to the mode change mechanism. On this event, an application switches to the road movement mode.

The whole application and the described mode change logic is modeled using component meta-model and its mode property extension in Figure 9.3 on the page 75.

9.3 Model Analysis

9.3.1 High-level Mode Set

Let us analyze the mode condition of the high-level mode set. It uses two boolean properties as input. The input properties are filled with data from two subcomponents of the component *Brain*. It is the component *InitBrain* which can indicate that the initialization is done and the component *WorldImage* which tells if the world has been completely searched and stored in memory. Assigning a value to the two indicators creates a set of events which affect the mode of the component *Brain*. It means that the mode is based on the state inside the component.

9.3.2 Low-level Mode Set

The most complex part of the model is inside the component *WorldSearch*. Its mode change mechanism contains a property function with three input properties and one of the properties is also its output property. It is the function computing a value of the property $mode_2$. Moreover, there is another property function which modifies a value of the property *townDiscDone* based on a value of the property $mode_2$. It creates a cycle and brings interesting behavior to the network.

Assume that the component *WorldSearch* is in the mode *TownDisc*, the robot is standing on the road or grass and the property *townDiscDone* is suddenly set to *true*. This event invokes a network reaction which causes that the property $mode_2$ is set to *RoadMov*. The reaction spreads back to the property *townDiscDone* and it is reset to *false*. This invokes the function which computes the property $mode_2$ again, but now the input values does not cause any output value modification and the network is stabilized. Thus the component is reconfigured to the mode *RoadMov* on this event.

Now assume the same situation but the ground the robot is standing on is a town. When the property *townDiscDone* is set to *true* the network reacts and the property $mode_2$ is set to *RoadMov*. It causes that the property *townDiscDone* is reset to *false* and the property $mode_2$ is computed again. Based on the input values which are now $groundType_3 = Town$ and $mode_2 = RoadMov$ the mechanism changes the value of the property $mode_2$ to *TownDisc* and the network is stabilized. It is the same property state as before the event occurred so the component is not reconfigured.

9.3.3 Evaluation

A property state machine generated from the network is partially shown in Figure 9.2. Most of property states corresponding to the initialization and standby mode are omitted because they do not bring any new insight. Total number of stable property states in PSM is 24. It is quite satisfying result considering the fact that there are 139 968 property states possible on 13 properties with the domains as specified.

An important aspect of the mechanism is its memory footprint. Each state, property and property value can be represented in memory by an integer number. The most complex part of the PSM representation is a transition function. It can be represented by a three-dimensional array – the first dimension to determine a state, the second and the third dimension to determine an event (a direct property and its value). Value of the array is an integer number referring to a target state. It means that the transition function

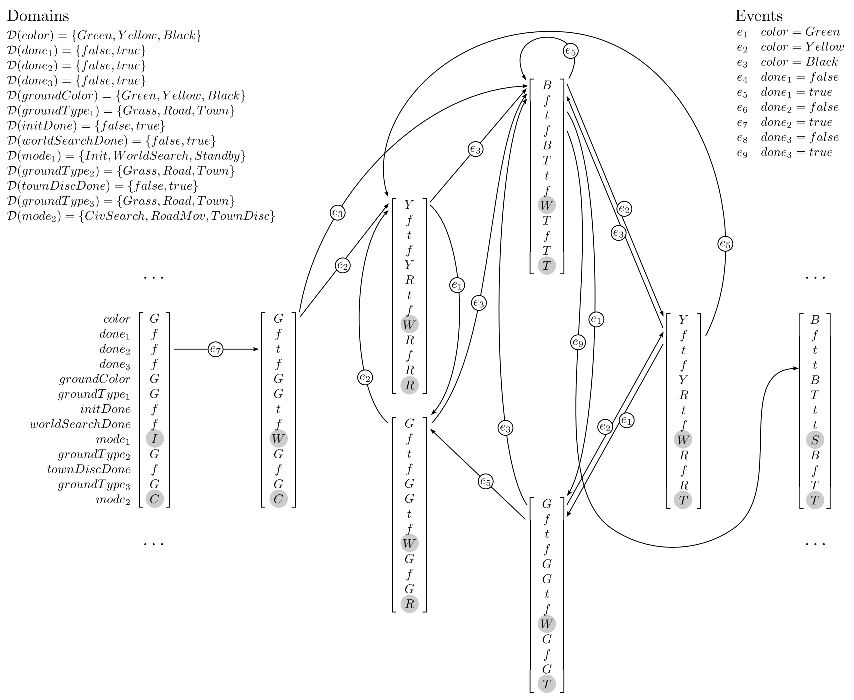

Domains
$\mathcal{D}(color) = \{Green, Yellow, Black\}$
$\mathcal{D}(done_1) = \{false, true\}$
$\mathcal{D}(done_2) = \{false, true\}$
$\mathcal{D}(done_3) = \{false, true\}$
$\mathcal{D}(groundColor) = \{Green, Yellow, Black\}$
$\mathcal{D}(groundType_1) = \{Grass, Road, Town\}$
$\mathcal{D}(initDone) = \{false, true\}$
$\mathcal{D}(worldSearchDone) = \{false, true\}$
$\mathcal{D}(mode_1) = \{Init, WorldSearch, Standby\}$
$\mathcal{D}(groundType_2) = \{Grass, Road, Town\}$
$\mathcal{D}(townDiscDone) = \{false, true\}$
$\mathcal{D}(groundType_3) = \{Grass, Road, Town\}$
$\mathcal{D}(mode_2) = \{CivSearch, RoadMov, TownDisc\}$
Events
e_1 $color = Green$
e_2 $color = Yellow$
e_3 $color = Black$
e_4 $done_1 = false$
e_5 $done_1 = true$
e_6 $done_2 = false$
e_7 $done_2 = true$
e_8 $done_3 = false$
e_9 $done_3 = true$
color
$done_1$
$done_2$
$done_3$
groundColor
$groundType_1$
initDone
worldSearchDone
$mode_1$
$groundType_2$
townDiscDone
$groundType_3$
$mode_2$

Figure 9.2: Part of the robot's property state machine

allocates:

$$\#states \ \cdot \ \#direct\ properties \ \cdot \ maximum\ domain \ \cdot \ sizeof(int)$$

$$(24 \cdot 4 \cdot 3 \cdot 4)\ Bytes = 1152\ Bytes$$

Finally, let us compare time complexity of the PSM-1 algorithm and the PSM-2 algorithm. Because of 9 possible events, time complexity of the PSM-1 algorithm is

$$24 \cdot 9 \cdot T_{react} = 216 \cdot T_{react}$$

In the PSM-2 algorithm, number of reaction calls is smaller. It can be estimated as follows. For the property *color*, let us call it an event property, there are 3 possible events – assigning the values *Green*, *Yellow* or *Black*. For this property, there is also the set of reactive properties $\{townDiscDone, mode_2\}$ which define 6 possible stable states. It means that there are 18 calls of the react algorithm for this event property. The same idea can be applied to the event properties $done_1$, $done_2$ and $done_3$ which define the following sets of reactive properties: $\{groundType_3, mode_2\}$, $\{worldSearchDone\}$ and $\{initDone\}$. It gives 18, 4 and 4 calls of the react algorithm. Thus the time complexity is:

$$(18 + 18 + 4 + 4) \cdot T_{react} \ + \ T_{second\ phase} = 44 \cdot T_{react} \ + \ T_{second\ phase}$$

9.4 Benefits

The case study shows that the component mode extension of the meta-model is meaningful. Component modes are encapsulated with their components and different mode sets are completely independent. Re-usability of the entire concept is preserved.

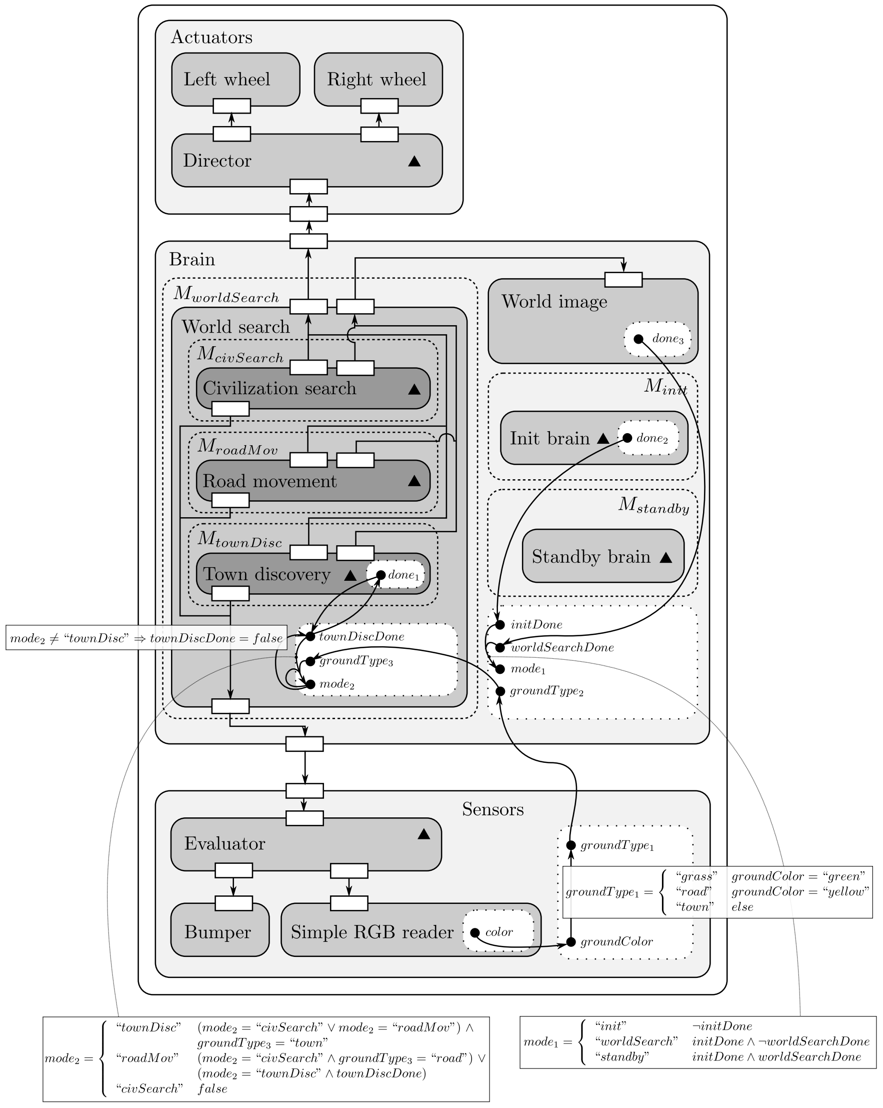

Actuators
Left wheel
Right wheel
Director
Brain
$M_{worldSearch}$
World search
$M_{civSearch}$
Civilization search
$M_{roadMov}$
Road movement
$M_{townDisc}$
Town discovery
$done_1$
World image
$done_3$
M_{init}
Init brain
$done_2$
$M_{standby}$
Standby brain
$mode_2 \neq$ "townDisc" $\Rightarrow$ townDiscDone = false
townDiscDone
$groundType_3$
$mode_2$
initDone
worldSearchDone
$mode_1$
$groundType_2$
Sensors
Evaluator
Bumper
Simple RGB reader
color
$groundType_1$
groundColor
$groundType_1 = \begin{cases} \text{"grass"} & groundColor = \text{"green"} \\ \text{"road"} & groundColor = \text{"yellow"} \\ \text{"town"} & else \end{cases}$
$mode_2 = \begin{cases} \text{"townDisc"} & (mode_2 = \text{"civSearch"} \vee mode_2 = \text{"roadMov"}) \wedge groundType_3 = \text{"town"} \\ \text{"roadMov"} & (mode_2 = \text{"civSearch"} \wedge groundType_3 = \text{"road"}) \vee (mode_2 = \text{"townDisc"} \wedge townDiscDone) \\ \text{"civSearch"} & false \end{cases}$
$mode_1 = \begin{cases} \text{"init"} & \neg initDone \\ \text{"worldSearch"} & initDone \wedge \neg worldSearchDone \\ \text{"standby"} & initDone \wedge worldSearchDone \end{cases}$

Figure 9.3: Robot's component model

The property based mode change mechanism works with several types of information, e.g. a color or boolean flags indicating a completed task, which are effectively represented by properties and distributed by property functions. The mode condition of the component $WorldSearch$ uses a possibility to compute a new active component mode on the basis of the current one. A current component mode is also used for adjusting a value of the property *townDiscDone*. All property functions – complex mode conditions, *groundColor* to *groundType* transformation and simple value propagations – are easily modeled using PFL.

After all, the application employs many features supported by the property based mode change mechanism which seems to be suitable for this kind of control systems.

Chapter 10

Related Work

This chapter provides a list of component-based frameworks aimed at real-time development which are relevant for the thesis. In this particular case, it means a support for some kind of runtime variability, mostly a support for dynamic reconfiguration. Exhaustive summary of real-time component frameworks can be found in [7].

10.1 MyCCM-HI

MyCCM-HI [14] is a component framework based on OMG Lightweight CCM. It is fully operational and ready to be used. The main idea of the framework lies in a transformation of an application designed by component-based approach into lower-level AADL language. A set of external tools like Cheddar and Ocarina can be used at the lower level to perform schedulability and deadlock analysis.

Reconfiguration is in MyCCM-HI based on operational modes [8]. Each component defines and implements a mode automaton that maintains an information about an active mode. The mode change mechanism consist of *boolean expressions* which decide about a mode transition, *messages* which can trigger a transition and *actions* which can be a result of a transition.

This is very similar to the property based mechanism introduced in the thesis. There is resemblance between boolean expressions defined in MyCCM-HI and mode conditions introduced in the thesis. Moreover, messages and activities are similar to properties and property functions defined in the property based mechanism as they manage distribution of mode-change relevant information between the component and its surrounding and adjust the inner state of the component.

The mode automaton of each MyCCM-HI component is transformed into a new control subcomponent that handles the parent component's reconfiguration. The control components are connected together with standard system's connection mechanisms in order to distribute messages.

This is the difference between MyCCM-HI and the property based mechanism. The property based mechanism generates control components as well but they are global for the entire application. It is possible because of the mechanism's better coupling with the system, especially because of the mechanism's abstraction to properties. Hence the mechanism can use an oracle based approach in reaction representation, while MyCCM-HI is forced to use its full simulation.

10.2 BlueArX

BlueArX [9] is a component framework aimed at traditional automotive control systems. It is developed by Bosh and fully operational.

Reconfiguration is based on operating modes. The framework focuses on design phase of the development cycle. It supports advanced development tools which helps with different analyses in different modes and even semi-automatic heuristic-based prediction of system's decomposition into modes. A modes define either different scheduling strategies, i.e components and processes are activated differently, or different control strategies, i.e. system determines different component configurations, different control signal paths and even different control path in source code. Development tools performing system analysis, e.g. WCET analysis, work with respect to different modes. At runtime, components are configured using *configuration interfaces*.

Understanding of modes in BlueArX and in the thesis is alike. Both define different configurations and different scheduling parameters. However, there is no explicit mode change mechanism at runtime in BlueArX. Reconfiguration at runtime has similarities with manual approach with support of the configuration interfaces. Advanced development tools analyze the whole reconfiguration mechanism in design phase and thus resolve the predictability issue.

10.3 Koala

Koala [10] is a component model aimed at consumer electronics development, created by Philips. The system attaches importance to optimization and managing diversity of target devices.

It supports partial dynamic reconfiguration at runtime. The reconfiguration is based on *switches* or *modules*, as a more general form of switches. Switches are units of composition which can reroute a connection between components at runtime on the basis of its dynamic configuration. The concept is very simple to realize and suitable for the targeted domain of devices.

However, it is not convenient for the domain of hard real-time systems because of the difficult analysis of real-time properties. There cannot be enabled and disabled different activities in different configurations as well. This can be a feature important for some control systems.

10.4 ROBOCOP

The Robust Component Model for Consumer Electronic Products (ROBOCOP) [11] is a component-based architecture for middleware layer in consumer electronics. It has been developed as an ITEA project. It consists of several frameworks. The core frameworks are development and runtime frameworks. The first one manages the development of components while the second one defines its execution environment. The core frameworks are supported by the download framework and the resource management framework.

Partial runtime variability is implemented by the download framework. It is able to download a component's update from the repository to the device and dynamically upgrade the component at runtime. However, it is not the true reconfiguration and comparison with the thesis is not possible.

Chapter 11

Conclusion and Future Work

The thesis has proposed an approach how to integrate a dynamic runtime reconfiguration support to a real-time application created by means of component-based development. The thesis has been focused on resolving high-level issues mostly. The basic meaning of operating modes in the context of component-based development has been discussed and a concept of a mode change mechanism has been introduced. One possible realization of the concept has been proposed using a concept of mode properties and functions. The property based mechanism has been analyzed in detail, implemented and a case study has been introduced as a proof of the concept. Because of the limited scope of the thesis, low-level aspects of the topic, i.e. a runtime architecture, has been described in less detail. It creates opportunity for further investigation.

First of all, more sophisticated mode change protocols have to be examined in the context of their possible application in the system, especially their ability to satisfy runtime structural requirements. Developing a new protocol fulfilling the requirements is also an option. Possibility to switch the mode change protocols according to situation is an optional feature.

A topic of mode change safety has not been discussed in the thesis. Safety mechanism have to be introduced in order to preserve unauthorized components from working with the system API methods which handle enabling and disabling application parts.

Another important topic of investigation is a non-oracle approach to a property network analysis. This method is closely related to a schedulability analysis. The analysis should incorporate a decision which approach is more suitable for a situation and a system should be able to choose between the two approaches accordingly.

Last but not least, a question of dynamic reconfiguration of distributed systems is open. Existence of a theoretical background that study deploying component-based real-time applications into distributed environment is crucial for the topic.

Appendix A

Content of the Enclosed DVD-ROM

The enclosed DVD-ROM is organised as follows:

- `master_thesis.pdf` – The thesis in PDF format.
- `readme.txt` – Content and usage description.
- `bin/` – Binary distributions of SOFA HI and Cushion.
- `src/` – Source code of the implementation.

Bibliography

[1] Hosek P.: *Supporting Real-time Features in a Hierarchical Component System*, Master Thesis, 2010

[2] Real J., Crespo A.: *Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal*, 2004

[3] Buttazzo G.C.: *Hard Real-Time Computing Systems*, ISBN 0-387-23137-4, 2005

[4] Borde E., Haik G., Watine V., Pautet L.: *Really Hard Time developing Hard Real Time*, Workshop Control Architecture of Robots 2007

[5] Masmano M., Ripoll I., Crespo A.: *Dynamic Storage Allocation for Real-Time Embedded Eystems*, 2003

[6] Crnkovic I., Larsson M.: *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, 2002

[7] Hosek P., Pop T., Bures T., Hnetynka P., Malohlava M.: *Comparison of Component Frameworks for Real-Time Embedded Systems*, 2010

[8] Borde E., Haik G., Pautet L.: *Mode-based Reconfiguration of Critical Software Component Architectures*, 2009

[9] Kim J.E., Rogalla O., Kramer S., Hamann A.: *Extracting, specifying and predicting software system properties in component based real-time embedded software development*, 2009

[10] Ommering R., Linden F., Kramer J., Magee J.: *The Koala Component Model for Consumer Electronics Software*, Computer 33, 78–85, 2000

[11] Maaskant H.: *A Robust Component Model for Consumer Electronic Products*, 2005

[12] Cerny O., Hosek P., Papez M., Remes V.: *SOFA 2 Component System documentation: User's and programmer's guide*, `http://sofa.ow2.org/docs/`

[13] Bruneton E., Coupaye T., Stefani J.B.: *The Fractal Component Model Specification*, `http://fractal.ow2.org/specification/`

[14] MyCCM High Integrity, `http://sourceforge.net/apps/trac/myccm-hi/wiki/`

[15] LEGO® MINDSTORMS®, `http://mindstorms.lego.com/`

[16] Extended BNF, ISO/IEC 14977, 1996

[17] OMG Unified Modeling Language™, Version 2.3, `http://www.uml.org/`

[18] Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf/`

[19] FreeRTOS, `http://www.freertos.org/`

[20] GNU/Linux, `http://www.linux.org/`