Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

Software Specification Refinement and Verification Method with I-Mathic Studio

Gerald H. HILDERINK
Imtech ICT Technical Systems,
P.O.Box 7111, 5605 JC, Eindhoven, The Netherlands
gerald.hilderink@imtech.nl

Abstract. A software design usually manifests a composition of software specifications. It consists of hierarchies of black box and white box specifications which are subject to refinement verification. Refinement verification is a model-checking process that proves the correctness of software specifications using formal methods. Although this is a powerful tool for developing reliable and robust software, the applied mathematics causes a serious gap between academics and software engineers. I-Mathic comprehends a software specification refinement and verification method and a supporting toolset, which aims at eliminating the gap through hiding the applied mathematics by practical modelling concepts. The model-checker FDR is used for refinement verification and detecting deadlocks and livelocks in software specifications. We have improved the method by incorporating CSP programming concepts into the specification language. These concepts make the method suitable for a broader class of safety-critical concurrent systems. The improved I-Mathic is illustrated in this paper.

Introduction

The proposed method is about formulating software specifications of components in a program. Software specifications aim at describing the desired behaviour of the software. Software specifications need to be as precise and complete as possible and should eliminates doubts about possible misbehaviours or uncertainties of the program. Incomplete software specifications in, for example, an embedded computer system can cause surprises that may have disastrous effects on its environment; e.g. a software crash in an airplane or rocket could risk people's lives, or defects in consumer electronics products due to a software bug could cause an economical disaster. We need tools and methods to get software specifications right and desirably the first time right.

A practical software specification method has been described by Broadfoot [1]. The method combines two formal methods, (1) the Cleanroom Software Engineering method [4] and (2) the CSP theory [3; 5]. The method allows for a precisely transformation of informal customer requirements into formal software specifications suitable for refinement verification by model-checking. The method has been developed on behalf of Imtech ICT Technical Systems in the Netherlands and it has been successfully applied to high-tech products at several technology leading companies, such as Assembléon and Philips Applied Technologies in the Netherlands. The method is called *I-Mathic*. The supporting toolset is called *I-Mathic Studio*.

After years of experience, we conclude that the method is suitable for relatively small controller components. This is because the method was originally based on a sequence-based modelling approach where by sequence enumerations can (easily) result in complex finite state machines. Nowadays, controller components in embedded systems get bigger and bigger due to a growing number of features they must perform. In order to overcome this problem, we investigated the use of CSP programming concepts to enrich the specification language and toolset with notion of concurrency. In general, concurrency is an important concept for dealing with the complexity of systems [2]. The result is an improved I-Mathic that is suitable for describing software specifications being scalable with the nature of complexity of computer systems. The new I-Mathic enhancements are introduced in this paper.

1. I-Mathic Modelling Approach

A complete composition of software specifications should describe the entire behaviour and operational structure of the program. It consists of hierarchies of abstract and concrete specifications which are subject to refinement verification. I-Mathic embraces a software specification refinement and verification method, which applies formal methods to precisely describe, analyse and connect separate software specifications in a systematic way. The I-Mathic method provides a practical and powerful approach for describing and verifying software specifications in an abstract, compositional, and provable manner.

1.1. Software Specifications and Requirements

A software specification of a software component manifests (a part of) the customer functional requirements or a particular part of the software design. A software specification is shaped such that it is suitable for software design. A software specification is expected to be a precise and complete description of the component and its interface.

The refinement and verification method assists the user in discovering and eliminating undefined issues in the customer requirements. The method helps with identifying additional requirements, so-called derived requirements.

An abstract specification is usually refined by a more concrete specification. In a hierarchy of specification a concrete specification can be an abstract specification of a deeper and more concrete specification.

Sometimes a concrete specification is called the implementation of the abstract specification. However, at the design phase an implementation is commonly described as a (most) concrete specification. Therefore, the proposed method treats a model of specifications. The term implementation is omitted until the implementation phase of the development process.

The abstract and concrete specifications, as a result of the proposed method, are computational and therefore they can be translated into code useful for simulation or model-checking. Of course, those specifications that are deterministic with stable states are useful to be translated into source code for program execution—these leaf specifications are the building blocks of the final program.

1.2. Applied Formal Methods

I-Mathic integrates four formal methods which provide a systematic and mathematical foundation. Each formal method takes care of a responsibility in de specification modelling process, as discussed below. I-Mathic Studio hides the mathematical foundation from the software engineer. This foundation is used by the toolset to provide the software engineer with automation and assistance during the modelling process. This results in consistent and complete software specifications. The following four[1] formal methods are incorporated:

- **Box Structured Development Method**. The box structured development method outlines a hierarchy of concerns by a box structure, which allows for divide, conquer and connect software specifications. The box structured development method originates from Cleanroom. Cleanroom defines three views of the software system, referred to as the black box, state box and clear box views. An initial black box is refined into a state box and then into a clear box. A clear box can be further refined into one or more black boxes or it closes a hierarchical branch as a leave box providing a control structure. This hierarchy of views allows for a stepwise refinement and verification as each view is derived from the previous. The clear box is verified for equivalence against its state box, and the state box against its black box. The box structure should specify all requirements for the component, so that no further specification is logically required to complete the component. We have slightly altered the box structured development method to make it more beneficial. This is discussed in Section 2.2.
- **Sequence-Based Specification Method**. The sequence-based specification method also originates from Cleanroom. The sequence-based specification method describes the causality between stimuli and responses using a sequence enumeration table. Sequence enumerations describe the responses of a process after accepting a history of stimuli. Every mapping of an input sequence to a response is justified by explicit reference to the informal specifications. The sequence-based specification method is applied to the black box and state box specifications. Each sequence enumeration can be tagged with a requirement reference. The tagged requirement maps a stimuli-response causality of the system to the customer or derived requirements.
- **CSP/FDR framework.** CSP stands for Communicating Sequential Processes [3; 5], which is a theory of programming. CSP is a process algebra comprising mathematical notations based on operators for describing patterns of parallel interactions. FDR is a model-checker for CSP. The CSP/FDR framework is used for formal verification. Box structures and sequence enumerations are translated to CSP algebra (in a machine-readable CSP format) which is input for FDR. A box structure of software specifications provides a hierarchy of refinement relationships between abstract and concrete specifications. These pairs of specifications can be verified for completeness and correctness using refinement checking. FDR also detects pathological problems in specifications, such as deadlock, livelock and race conditions.
- **CSP Programming Concepts**. The CSP programming concepts implement a subset of the CSP theory by software engineering constructs. These pragmatic constructs capture concurrency at a high level of abstraction (far beyond multi-threading) and hide the CSP mathematics from the user. These CSP programming concepts can be build into a programming language (e.g. occam, Ada, Limbo, Handel-C, HASTE), provided by libraries (e.g. CP, JCSP, C++CSP, CTJ), or being part of a specification language (e.g. gCSP, I-Mathic). These concurrency concepts are often seen as a welcome contribution to the user's knowledge of sequential programming and modelling. These concepts can also be used for programming hardware.

[1] The method described by Broadfoot [1] includes only the first three formal methods.

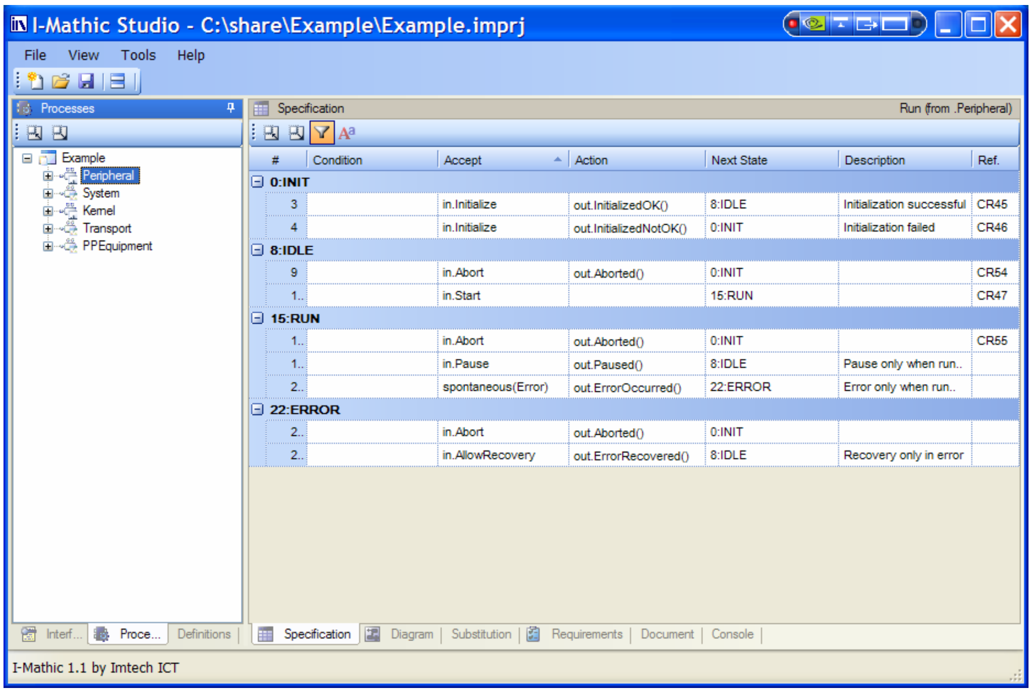
The black box, state box and clear box views are distinct usage perspectives which are effective in defining the behaviour of individual components but they provide little in the way of compositionality. Combining specifications cannot make statements about the behaviour as a whole [1]. Cleanroom provides no means of describing concurrent (dynamic) behaviours and analysing pathological problems, such as deadlock, livelocks, race conditions and so on.

In Cleanroom, notion of concurrency was occasionally mentioned as a solution to deal with describing control structures, but no sound semantics was given. Since Cleanroom is sequence-based, concurrency adds a new dimension to the complexity of the sequence-based specifications. We argue that the use of concurrency should make the specifications of complex systems simpler and natural rather than more complex and artificial. In fact, concurrency provides compositionality. Sound semantics of concurrency is given by CSP. The lack of concurrency and compositionality in Cleanroom can be solved by applying CSP programming concepts to box-structures and sequence enumerations. This is why the CSP programming concepts are integrated in I-Mathic, which affect the Cleanroom box structured development method and the sequence-based specification method.

The CSP programming concepts and the CSP theory are considered two different formal methods with respect to the different contexts they are intended for.

1.3. I-Mathic Supporting Toolset – I-Mathic Studio

The I-Mathic method is supported by a software development toolset, which is called I-Mathic Studio. Figure 1 shows a screenshot of the graphical user interface. The tree view on the left side is used for navigation and adding/removing/renaming specification elements. The tree view is divided in three sections: service interfaces, process classes and definitions. Service interfaces and process classes are discussed in Sections 2.2.1. The definitions are global enumeration types and object types that are used by the elements. The panel on the right side is used to model the specifications or diagrams. The example in Section 4 shows more screenshots.



#	Condition	Accept	Action	Next State	Description	Ref.
0:INIT						
3		in.Initialize	out.InitializedOK()	8:IDLE	Initialization successful	CR45
4		in.Initialize	out.InitializedNotOK()	0:INIT	Initialization failed	CR46
8:IDLE						
9		in.Abort	out.Aborted()	0:INIT		CR54
1..		in.Start		15:RUN		CR47
15:RUN						
1..		in.Abort	out.Aborted()	0:INIT		CR55
1..		in.Pause	out.Paused()	8:IDLE	Pause only when run..	
2..		spontaneous(Error)	out.ErrorOccurred()	22:ERROR	Error only when run..	
22:ERROR						
2..		in.Abort	out.Aborted()	0:INIT		
2..		in.AllowRecovery	out.ErrorRecovered()	8:IDLE	Recovery only in error	

Figure 1. I-Mathic Studio graphical user interface.

The toolset assists the user with the specification process and it automates the following features:

- Create/load, edit, refactor, and save software specifications,
- Navigation between the software specifications,
- Specifying relationships between software specifications, such as equivalence relationships and communication relationships.
- Transformation of the software specifications to machine-readable CSP for input for the FDR model-checker,
- Depicting software specifications by state transition diagrams or process diagrams,
- Code generation to, for example, C++,
- Document generation, e.g. derived requirements overview.

The code generator is customizable to the customer's coding standards and programming language. Most specification elements can be documented separately and tagged with a requirement reference. This allows for requirements traceability. The software specifications are saved in XML format, which can be used by other software design tools. I-Mathic Studio is evolving with additional features, which are not mentioned in this paper.

2. Process-oriented Software Specification Development

2.1. Concurrency

The inputs and outputs of a computer system interact with a concurrent world. The inputs (stimuli) and outputs (responses) of a program synchronize with the outside world and that perform in sequence, in parallel or by some choice. It is important to be able to describe the behavioural relationships between the concurrent inputs and outputs. Not surprisingly, describing parallel inputs and outputs by a single sequence enumeration is far too complex. Therefore, the software specification refinement and verification method should encompass concurrent concepts. Concurrent concepts are provided by the CSP theory.

Describing the behaviour of a complex process with sequential enumerations can easily result in a complex description. The vast number of states and transitions makes reasoning and modelling difficult. Describing concurrency aspects (e.g. synchronization, parallelism, non-determinism, and timing) in a finite state machine (being inherently sequential) will make it even harder or perhaps impossible. Therefore, concurrency concepts are required in order to manage the complexity of components. Therefore a software specification should be able to describe a composition of smaller and simpler specifications.

A software specification can be described by a single process or by a network of parallel processes. A network of processes is a process itself (networks of networks).

Both single process and network descriptions can reflect deterministic and non-deterministic behaviours. Non-determinism is a natural phenomenon in a system's behaviour, which should not be ignored. The granularity of (non-)determinism depends on the abstraction of the process. The ability to describe non-determinism can make a specification shorter, more natural, and more abstract. The interface of a process reveals the behaviour of interaction via its ports.

2.2. Box Structured Development Method in the light of processes

In Cleanroom, the box structured development method defines three views of the software system, known as black box, state box and clear box views. A box structure consists of usage components (boxes) in the software development method which results in a complete mapping between the customer requirements and the structural operations of the resulting software. However, the sequential nature of the sequence-based specification method and the lack of composing behaviours limit the scalability of the complexity. Hence, Cleanroom is useful for small components. We argue that box structures in the light of communicating processes will eliminate this limitation.

In I-Mathic, a box represents a process. A process performs a life on its own and it is completely in control of its own data and operations which are encapsulated by the process. A process can play the role of both a usage component *and* a building block of the system architecture. I-Mathic enables a straightforward mapping between a software specification and its implementation.

2.2.1. Black Box and White Box Specifications

Using the Cleanroom's box structures, we experienced a strong coupling between a black box view and state box view, which results in a single software specification. It is more efficient to define an abstract specification and a concrete specification for respectively the outside and inside of a process. Therefore, two abstract views of a process are defined which refer to as the black box and white box views of a process. These two views replace the three views in Cleanroom. A black box view describes an abstract specification and a white box view describes a concrete specification of a process, respectively called a *black box specification* and a *white box specification*. A black box specification treats the system as a "black-box", so it doesn't explicitly use knowledge of the internal structure. Black box specification focuses on the functional requirements. White box specification peeks inside the "box", and it focuses specifically on using internal knowledge (design) of the process. The white box specification describes a software specification at a lower level in hierarchy. It does not describe the implementation of the process. The black box and white box views/specifications can be composed of concurrent sub-specifications. The black box and white box specifications of a process share the same process interface (or a subset). Both the black box and white box specifications can be described by a sequence enumeration (Section 2.3) or by a network of processes.

A black box specification can be created at any time. A black-box specification is only necessary when a process design must be tested against the requirements. This is usually the top process, but it can be necessary to specify black box specifications at other levels in the design of which the requirements pass judgment.

The specifications at the leaf ends of the box structure are typically black box specifications. They describe the behaviour of custom or third-party code.

The black box and white box specifications of a process share the same process interface. Both specifications share a set (all or a subset) of ports of the process interface. The white box specification of a process can be a refinement of one or more back box specifications (if given). Each black box specification describes a particular client view of the process. If a process provides both black box and white box specifications then both behaviours, with respect to the shared set of ports, must be equal!

This process-oriented box structured development method is effective in detecting and identifying defects in the customer requirements. Furthermore, the composition of processes renders a communicating process architecture, which organizes structural and functional decomposition.

2.2.2. Process-Oriented and Service-Oriented Design

Two types of classifications define the process-oriented and service-oriented design aspects of software specification, namely:

- **Process Classes**. A process is an instance of its process class. Process classes define the type of the processes. A process class defines a process interface and describes the behaviour of the interface. Two types of process classes are distinguished: (1) sequence enumeration process classes and (2) network process classes.
- **Service Interfaces**. A service interface is a collection of services a process can offer. A process class must implement a service interface in order to perform its services. Services are implemented as methods. Service interfaces define the type of the ports and channels. A process (that implement the service interface).

A process class defines public or private methods. Public methods implement services they can accept via their input ports. Private methods can be invoked by the sequence enumeration table, but are invisible to other processes. Methods are described by sub-sequence enumeration tables, which create a hierarchical finite state machine.

Processes specify distinguishable behaviours at a high level of abstraction—higher than objects. Hence, software specifications are described by non-objects, namely:

- *processes* that provide the building blocks of a software architecture, which compose functional tasks, responsibilities and concerns,
- *services* that processes can offer,
- *ports* that specify the process interface,
- *channels* that connect processes via ports,
- and *sequence enumerations* that describe the protocol of interaction in terms of events, states, decisions and transitions.

The communication relationships between processes are defined by channels that connect the process interfaces. See Figure 2. The communication relationship determines the integrity and consistency of a network of software specifications.

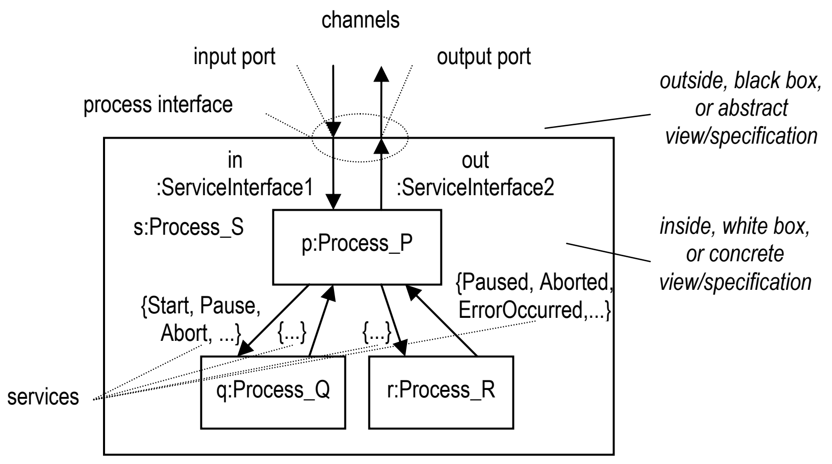

channels
input port
output port
process interface
outside, black box, or abstract view/specification
in :ServiceInterface1
out :ServiceInterface2
s:Process_S
p:Process_P
inside, white box, or concrete view/specification
{Paused, Aborted, ErrorOccurred,...}
{Start, Pause, Abort, ...}
{...}
{...}
services
q:Process_Q
r:Process_R

Figure 2. Box Structure of a single process consisting of multiple inner processes.

A process is identified by a process name and a process class. The process class defines the type of the process; i.e. the process interface and the software specification.

The ports are identified by a port name, service interface and a direction. The service interface and the direction define the port type. The services a process can offer are bound by the set of input ports. Each service interface defines a view or group of services a client process can request. Input ports can be connected output ports, or an input/output port can be connected to its parent input/output port. In either ways, the port types must be equal.

The connections between processes are channel. Channels take care of synchronisation, process scheduling and data transfer. Channels are initially unbuffered and perform handshaking between interacting processes. Buffered channels are optional in circumstances of structural performance bottlenecks [2].

2.3. Sequence Enumerations in the light of processes

A sequence enumeration describes the possible sequences of stimuli, responses and states. The stimuli involve the inputs and the responses involve the outputs of a process. A stimulus is an incoming event, i.e. it accepts and receives a message. The stimulus triggers a transition from one state to another state. The transition performs a response before the next state. A response is an outgoing event, i.e. sending a message. The states, internal actions and the state transitions are completely encapsulated by the process.

Table 1 illustrates a sequence enumeration table. The layout of a sequence enumeration table consists of states and a number of transitions (entries) per state. A transition describes a branch which is part of a sequence enumeration. A transition is ready to be executed when the condition and accept fields are both true. A sequence enumeration waits in a current state until at least one transition becomes ready. In case more than one transition is ready then one of these transitions will be randomly[2] selected and executed. On selection, the accept and action fields are executed and then a next state is taken.

Table 1. Sequence Enumeration Table Layout.

state 1					
condition	*accept*	*action*	*next state*	*description*	*ref.*
…	…	…	…	…	…
…	…	…	…	…	…
…	…	…	…	…	…
state 2					
condition	*accept*	*action*	*next state*	*description*	*ref.*
…	…	…	…	…	…
…	…	…	…	…	…
…	…	…	…	…	…

The fields are described as follows:

- **Condition field**. The condition is a Boolean expression using state variables and Boolean operators (currently only '==' and '!').
- **Accept field**. The accept field specifies a stimulus, i.e. an input port and a service, which is ready when another process is willing to request the service via the port. If the condition is true, the accept field is ready *and* the transition is selected then the *service is accepted*. The service is executed by the process as part of the acceptance.

[2] The semantics of the choice construct is derived from the external choice construct in CSP.

- **Action field**. The action field specifies zero, one or more sequential actions. The actions are restricted to sending responses and updating internal state variables. This field is like a small code body with a C-like syntax. A response is described by an output port and a service. A response is a request of a service via the output port to a server process. The response only is executed by the server process on acceptance by the server process.
- **Next State field**. The sequence enumeration takes the next state immediately after the action field has terminated
- **Description field**. A transition can be described separately. This field is optional.
- **Reference field**. The reference field specifies a requirement tag, which enables requirement traceability. A transition can refer to a customer requirement or a derived requirement. This field is optional.

The sequence enumeration tables define the client-server relationships between processes. The accept field defines the server role of the process and the response in the action field defines the client role of the process. Examples of sequence enumeration tables are given in Section 4.

3. Refinement Verification

A specification that is equivalent to a simpler specification can be replaced by the simpler one. Let process *P* be a complex white box specification and process *Q* a black box specification. Process *P* was developed from a design perspective and process *Q* from a requirement perspective. The processes *P* and *Q* must share the same interface; i.e. refinement is only possible if the number of ports, their types and order are equal. FDR will check if *P* is trace-failure-divergence compatible with *Q*, i.e. the future stimuli and responses of *P* and *Q* must be the same. In case they are not trace-failure-divergence equivalent, *P* is not a refinement of *Q*. This indicates incomplete requirements or a mapping failure between the informal and the formal specifications. In this case, iterations with the customer are required that must clarify and solve the mismatch.

Abstract software specifications usually represent the requirements, which are called *predicates*. A predicate specifies properties of a process that must hold.

The refinement check in CSP is given by:

$$\text{Spec'} \sqsubseteq \text{Spec}$$

Spec' is the predicate being asserted and *Spec* is tested against *Spec'*. *Spec* passes just when all its behaviours are one of *Spec'*. In other words, the behaviours of *Spec'* include all those of *Spec*. *Spec* is a refinement of *Spec'*. The refinement-check considers all the process' behaviours, namely traces, failures and divergences.

In case *Spec'* and *Spec* are both predicates, their behaviours must be equivalent, so:

$$\text{Spec'} \sqsubseteq \text{Spec} \quad \text{and} \quad \text{Spec} \sqsubseteq \text{Spec'}$$

should hold. This is called the equivalence relationship between two specifications:

$$P \text{ is equivalent to } Q$$

If *P* is equivalent to *Q* and *Q* is a simpler description then *Q* can be used instead of *P* for further analysis. This simplifies and accelerates the refinement verification in the hierarchy of specifications.

In I-Mathic, a specification can be verified for equivalence with another specification; no matter if this is a black box or white box specification. Verification is possible as long as the process interfaces are compatible.

4. Example

An example of a simple system is illustrated in this section. We do not have the space to discuss the system in detail. The exact meaning of the processes, their detailed specifications and the requirements are omitted. The example illustrates the look-and-feel of the I-Mathic method and I-Mathic Studio. A complete illustration of I-Mathic is provided by an I-Mathic course at Imtech ICT.

The example comprehends the interaction between a user and an embedded system. The user interacts with the system via a graphical user interface (GUI). The embedded system is a system controller consisting of three subsystems; a supervisory controller and two peripheral controllers. See Figure 3. The kernel controller performs the supervisory control over the transport and pick & place controllers. The transport and pick & place controllers contain other subsystems. These subsystems are omitted in this paper. Each controller is a process that is connected via channels; see the arrows in the Figure. The focus of this example is on the embedded system and not on the GUI.

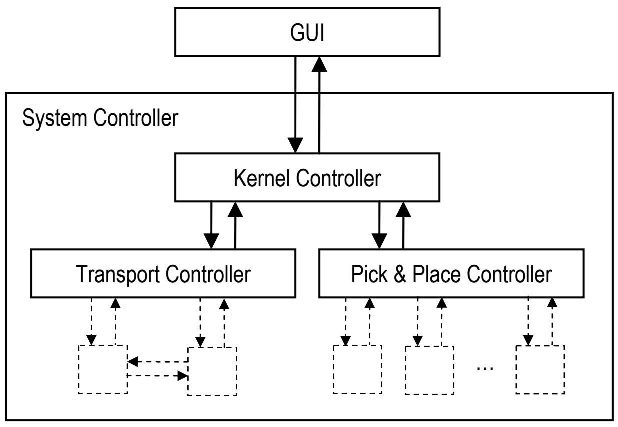


Figure 3. Process architecture of controllers.

The process names and process classes are given in Figure 4. The process class System describes a white box specification by a network of processes, since we peek inside System. The Kernel process class is the specification that needs to be developed.

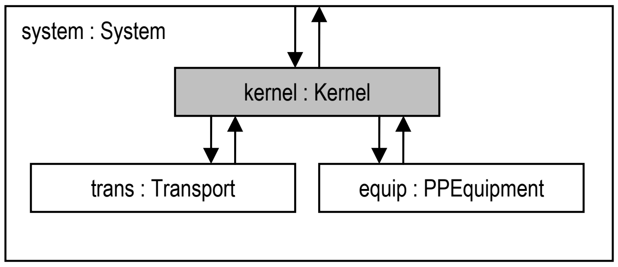


Figure 4. Network of processes with identifiers.

Coincidentally, the system, trans and equip processes have the same interface *and* they must behave similar. Hence, they have the same black box specifications. The black box specification is defined by the Peripheral process class. We must prove that the process classes System, Transport and PPEquipment are equivalent to Peripheral. See Figure 5.

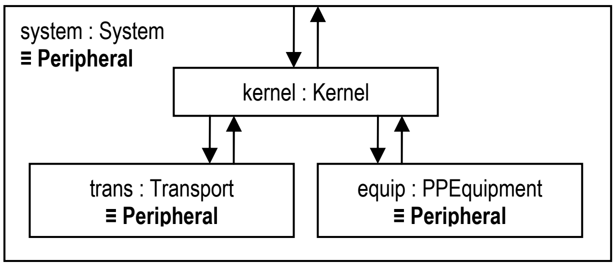
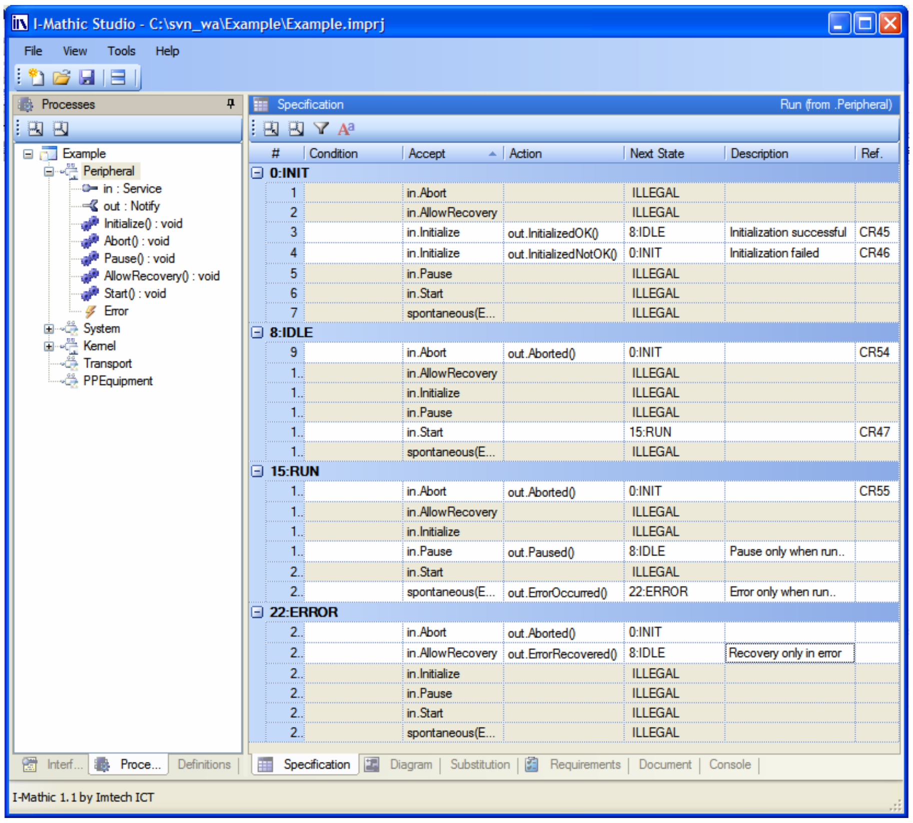


Figure 5. Equivalence relationships specifying refinement verifications.

Since Transport and PPEquipment are customer-made components, they have no white box specification. These components needs to be tested using test cases. In this example, we assume that Transport and PPEquipement are equivalent to Peripheral. The Kernel specification must be developed such that System fulfils the functional requirements and that System is equivalent to the Peripheral specification.

In case System, Transport and PPEquipment are equivalent to Peripheral then Peripheral can be used instead for further refinement verification. This optimizes the refinement verification. FDR will use less state space and the model-checking will perform faster.

The Peripheral specification is given in I-Mathic Studio in Figure 6. The Peripheral process class is selected in the tree view. The expanded Peripheral process class shows the ports, services it can offer and an internal event (Error).



#	Condition	Accept	Action	Next State	Description	Ref.
0:INIT						
1		in.Abort		ILLEGAL		
2		in.AllowRecovery		ILLEGAL		
3		in.Initialize	out.InitializedOK()	8:IDLE	Initialization successful	CR45
4		in.Initialize	out.InitializedNotOK()	0:INIT	Initialization failed	CR46
5		in.Pause		ILLEGAL		
6		in.Start		ILLEGAL		
7		spontaneous(E...		ILLEGAL		
8:IDLE						
9		in.Abort	out.Aborted()	0:INIT		CR54
1..		in.AllowRecovery		ILLEGAL		
1..		in.Initialize		ILLEGAL		
1..		in.Pause		ILLEGAL		
1..		in.Start		15:RUN		CR47
1..		spontaneous(E...		ILLEGAL		
15:RUN						
1..		in.Abort	out.Aborted()	0:INIT		CR55
1..		in.AllowRecovery		ILLEGAL		
1..		in.Initialize		ILLEGAL		
1..		in.Pause	out.Paused()	8:IDLE	Pause only when run..	
2..		in.Start		ILLEGAL		
2..		spontaneous(E...	out.ErrorOccurred()	22:ERROR	Error only when run..	
22:ERROR						
2..		in.Abort	out.Aborted()	0:INIT		
2..		in.AllowRecovery	out.ErrorRecovered()	8:IDLE	Recovery only in error	
2..		in.Initialize		ILLEGAL		
2..		in.Pause		ILLEGAL		
2..		in.Start		ILLEGAL		
2..		spontaneous(E...		ILLEGAL		

Figure 6. Peripheral process class in I-Mathic Studio.

The sequence enumeration table on the right side shows the input ports (and services) and internal event (spontaneous (Error)) in the accept fields for each state. The internal event can occur inside the process (like a spontaneous exception) and therefore it is not passed via the process interface.

The white rows are specified transitions going to a next state. The grey rows are so-called *ILLEGAL transitions*. ILLEGAL transitions are supposed not to happen. The model-checker will notify the user if there is a possibility that the process can end in an ILLEGAL state, due to an illegal service request. Thus, illegal transitions are detected during refinement verification and not at run-time. This saves time and the method guarantees that all illegal requests are detected and solved at the early specification phase of the development.

For each sequence enumeration table a state transition diagram (STD) can be shown. An STD gives an overview of states, transitions and a sequential thread of control. The ILLEGAL states are not shown in an STD because this would make the STD unnecessary complex. Figure 7 shows the STD of the Peripheral specification.

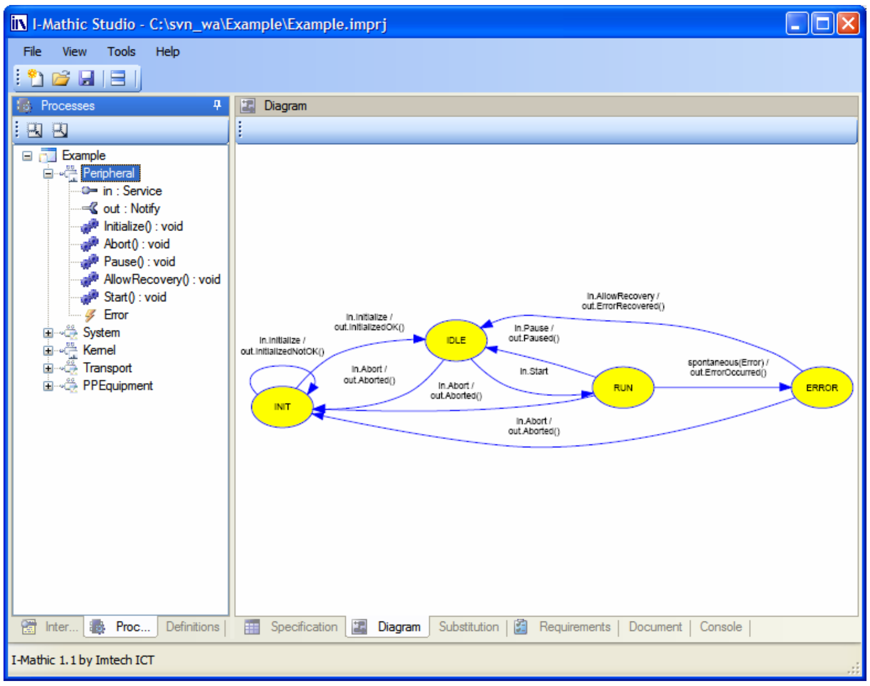
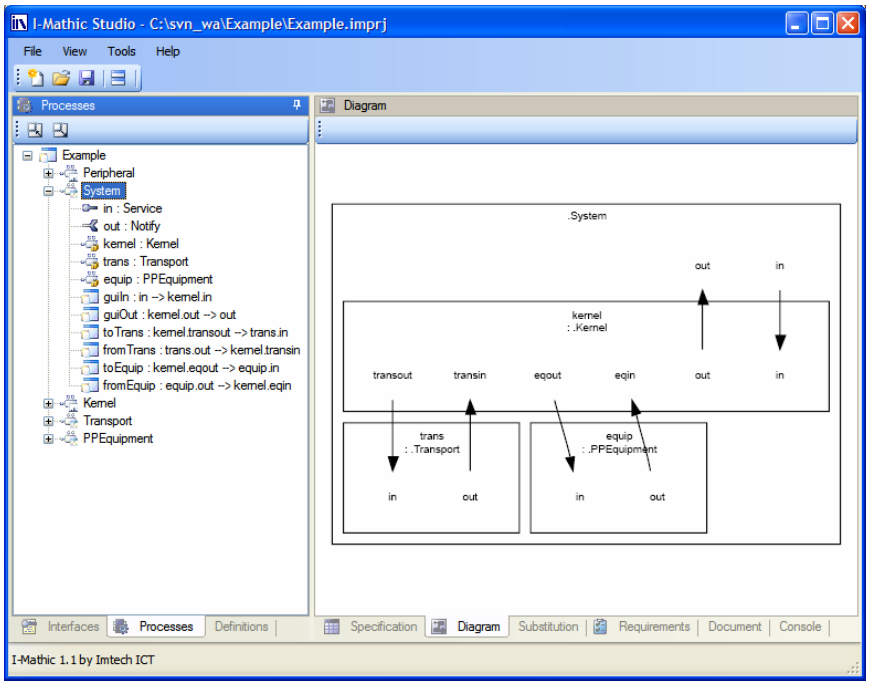


Figure 7. State transition diagram of Peripheral process class.

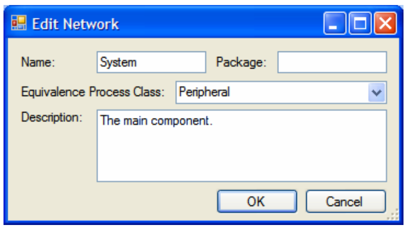
The System process class describes a network of processes, which can be depicted by a process diagram (PD). Figure 8 shows the network of processes and connections. The System process class is clearly a white box specification. A PD can be shown at every level in the process hierarchy.

A concrete specification can be tested for equivalence by specifying an abstract specification. For example Figure 9 shows the properties of the System process class. Here, Peripheral is specified as the abstract specification. Therefore, the System process class will be verified against the Peripheral process class. This is also done for the Transport and PPEquipment process classes. The Kernel process class does not have an equivalence relationship specified and therefore kernel will only be part of the System specification.


I-Mathic Studio - C:\svn_wa\Example\Example.imprj
File View Tools Help
Processes
Diagram
.System
out
in
kernel
: .Kernel
transout
transin
eqout
eqin
out
in
trans
: .Transport
in
out
equip
: .PPEquipment
in
out
Interfaces Processes Definitions
Specification Diagram Substitution Requirements Document Console
I-Mathic 1.1 by Imtech ICT

- Example
  - Peripheral
  - System
    - in : Service
    - out : Notify
    - kernel : Kernel
    - trans : Transport
    - equip : PPEquipment
    - guiIn : in -> kernel.in
    - guiOut : kernel.out -> out
    - toTrans : kernel.transout -> trans.in
    - fromTrans : trans.out -> kernel.transin
    - toEquip : kernel.eqout -> equip.in
    - fromEquip : equip.out -> kernel.eqin
  - Kernel
  - Transport
  - PPEquipment

Figure 8. Process diagram of Kernel process class.



Name: System
Package:
Equivalence Process Class: Peripheral
Description: The main component.

Figure 9. System should be equivalent to Peripheral.

FDR checks the model successfully. Each equivalence relationship is true and the system is deadlock free. Unfortunately, we did not succeed to make a screenshot in time of the Linux machine on which FDR runs. The results of FDR are not further discussed.

5. Conclusions

The method allows the user to describe software specifications in terms of processes. A process can be described by a sequence enumerations table or by a network of parallel processes. This process-oriented framework allows the user to compose a complex system by simpler sub-systems. The absence of objects allows the user to reason clearly about the functional behaviour of the system and not being disturbed by the limitations of (inherently sequential) object-oriented structures. Eventually, object-oriented and multithreaded structures in the implementation are determined by the process architecture.

The software specification refinement and verification method has been improved with CSP programming concepts. These concepts provide the user with practice software engineering constructs to describe concurrent behaviours in software specifications. I-Mathic includes sequential, parallel and choice constructs, and channels that synchronize parallel sequence enumerations (processes). These concepts are required for the compositionality and scalability of software specifications.

The CSP/FDR framework allows for refinement verification between abstract (black box) and concrete (white box) software specifications. This refinement verification process proves the completeness and correctness of the software specifications. Also, deadlock, livelocks and race conditions are checked.

I-Mathic Studio has a graphical user interface which includes features that assist the user with the development process of software specifications. This improves the productivity and the user does not have to an expert in formal methods.

6. Future work

The CSP programming paradigm provides I-Mathic a road map of extra new features that are useful for describing software specifications of complex and mission-critical embedded systems. A list of additional features, such as debugging in I-Mathic Studio (rather than in FDR), animation of execution, colouring deadlocks and livelocks in sequence enumeration tables and graphical modelling are planned to be implemented in I-Mathic Studio.

References

[1] Broadfoot, G. H. (2005). "Introducing Formal Methods into Industrial using Cleanroom and CSP." Dedicated Systems Magazine Q1.

[2] Hilderink, G. H. (2005). Managing Complexity of Control Systems through Concurrency. Control Laboratory. Enschede, The Netherlands, University of Twente.

[3] Hoare, C. A. R. (1985). Communicating Sequential Processes, Prentice Hall.

[4] Prowell, S. J., C. J. Trammell, et al. (1998). Cleanroom Software Engineering - Technology and Process, Addison-Wesley.

[5] Roscoe, A. W. (1998). The Theory and Practice of Concurrency, Prentice Hall.