

A Case Study in Requirement Analysis of Control Systems using UML and B¹

Colin Snook

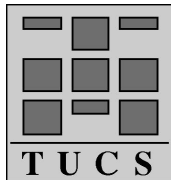
University of Southampton, Department of Electronics and Computer Science, SO17 1BJ, UK

Leonidas Tsiopoulos,
Marina Waldén²

Åbo Akademi University, Department of Computer Science,
Lemminkäisenkatu 14, FIN-20520 Turku, Finland

¹ Work done within the MATISSE-project, IST-1999-11435,
<http://www.matisse.qinetiq.com/>

² Financed by the Academy of Finland



Turku Centre for Computer Science
TUCS Technical Report No 533
June 2003
ISBN 952-12-1174-1
ISSN 1239-1891

Abstract

When developing critical systems that should be highly dependable we need to ensure that these systems satisfy functional as well as safety requirements. This can be achieved by developing the system with a formal method in a stepwise manner and proving the correctness of each step. For industrial strength systems, though, we need a graphical interface to the formal method. In this paper we develop part of a microplate liquid handling workstation, where we express the informal requirements and the refinements in UML. We translate the UML diagrams to B-action systems with the tool U2B. In the B-action system formalism we then prove the correctness of the development using the provers of the tool Atelier B.

Keywords: Formal Methods, Industrial Case Study, The B Method, Action Systems, UML, Tool Support

TUCS Laboratory
Distributed Systems Laboratory

1. Introduction

In this paper we illustrate a method for specifying and verifying the behaviour of a fault tolerant control system that is part of a safety critical healthcare system. The method allows failure recovery behaviour (as well as normal functionality) to be specified in increasing detail by stepwise superposition refinements [BaKu83, Katz93]. At each refinement the consistency of the specification with its preceding abstraction is formally verified. The method uses a visual notation based upon a representation of action systems in the UML. An automatic translation tool [SB00] is used to generate a textual form of the specification suitable for verification using existing tools. The case study used to illustrate the method is a microplate liquid handling workstation [PE01] and we focus on the part of the system that supplies the workstation with microplates. A full version of the case study is described elsewhere [Tsiopoulos03]. When developing safety critical systems we need to ensure that functional as well as safety requirements are satisfied [Troubitsyna00].

We propose a method [PTW02, PTWBEJ01] where we depict the informal requirements and the safety properties of the system with UML diagrams [PS00]. In order to prove that the safety properties are ensured and that the specification is consistent we translate the UML specification into B-action systems [BW98, WS98]. The translation is done with the tool U2B [SB00]. B-action systems is a formalism for supporting the development of complex distributed systems. The B Method [Abrial96] and its associated tool Atelier B [Steria96] provide us with a good mechanised support for the consistency proof of the B-action systems.

Using superposition refinement we stepwise add more functionality to the specification and turn it into a more concrete and deterministic system incorporating more concrete safety aspects throughout the process. We model the new features of the system by refining the class- and statechart diagrams. For each step the diagrams are translated with the U2B translator to B-action systems. The refined B-action systems are proved using the provers of Atelier B. During the refinement procedure the system is transformed into a control system consisting of a collection of plant, controller, sensors and actuators [PRWJ01].

The UML class- and statechart diagrams provide a graphical interface to the formal development process. Furthermore, we get a consistent documentation of every stage of the development process. Static as well as dynamic behaviour is described in the class- and statechart diagrams. The U2B translator then combines these behaviour descriptions into B machines.

We incorporate safety analysis throughout the software development process [PTWBEJ01, Troubitsyna00] to achieve the required dependability. The safety analysis starts by identifying hazards that are potentially dangerous in the abstract specification. While stepwise refining the system we produce more concrete descriptions of the hazards and find means to deal with them. We model fault occurrence and detection, as well as how the system behaves in the presence of faults. The system should behave in a predictable way also in the presence of failures.

We proceed by describing the case study in Section 2. In Section 3 we describe how the abstract model is created. The refinement process of the model is then depicted in Section 4. The case study is used to exemplify all stages of the development. We conclude in Section 5.

2. The Fillwell case study

The system being developed as a case study in this paper is part of a product of PerkinElmer's – Fillwell™, a microplate liquid handling workstation, preparing samples [PE01]. PerkinElmer Life Sciences designs, manufactures, and develops analytical systems for use in drug discovery, research, and other bioresearch and clinical diagnostics areas. The manufactured analytical systems include reagents, sample handling and measuring instrumentation, as well as computer software. The Fillwell workstation was the first liquid handling system designed for high density microplates. The system is safety-critical. The direct harm to the humans using the drug discovery systems is quite moderate according to the classification for normal safety-critical systems. However, the indirect harm caused by the results of incorrectly performed experiments might be catastrophic.

The Fillwell system belongs to the class of products for drug discovery and bioresearch. It consists of a base unit and a rotary unit that are attached as shown in Figure 1. The Dispense Head in the base unit dispenses liquid into microplates on an Operating Table, as well as on the Rotary Table in the rotary unit. The Operating Table contains three plate positions while the Rotary Table has six plate positions. The gantry can move the Dispense Head horizontally and vertically with a very high precision over all the positions on the Operating Table and the processing position on the Rotary Table. There are also four Stackers in the rotary unit above the Rotary Table providing plates for the Rotary Table. Each of the Stackers can place a plate on and remove a plate from the Rotary Table. In this case study we focus on the Stackers of the Fillwell system. The complete rotary unit and the base unit are documented elsewhere [Tsiopoulos03, BJW03].

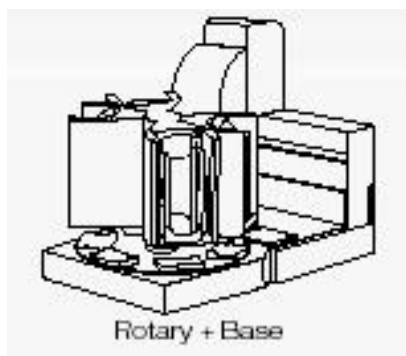


Figure 1. The Fillwell workstation [PE01].

2.1 The requirements of the case study

In this case study we focus on the stackers in the rotary unit of the Fillwell workstation. The Stackers of the Fillwell workstation are placed over the Rotary Table. The Rotary Table has six positions with a *plate holder* in each position. The position/sector closest to the Operating Table is called the *processing position* and is considered as position 1. The four Stackers are above the positions 2, 3, 5, and 6. The top view of the Rotary Table and the Operating Table is given in Figure 2.

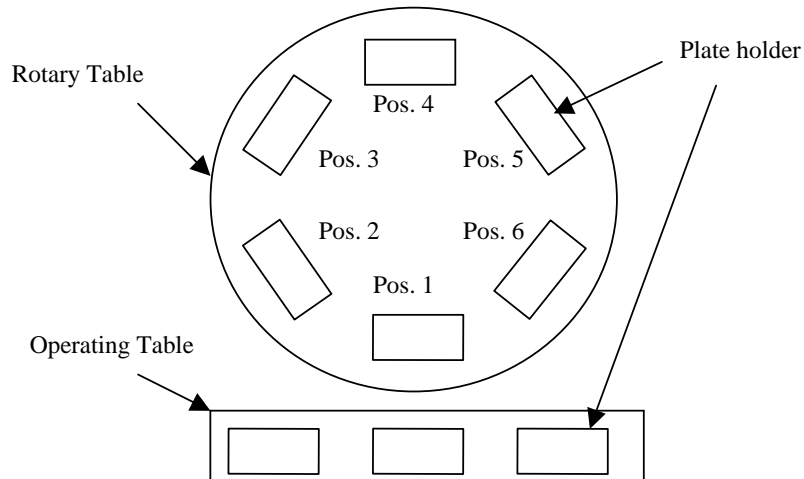


Figure 2. Top view of the Rotary Table.

Each stacker has a maximum capacity of twenty plates. The task of a stacker is to *destack* (place a plate on the Rotary Table) and to *stack* (remove a plate from the Rotary Table). The stackers have *arms* to hold the plates and to stack or destack them. Initially, some stackers can be *full* while others can be *empty*.

The following safety requirements guarantee that the arms of the Stackers will not collide with the Rotary Table:

- the arms should not be extended when the Rotary Table is rotating,
- when a Stacker is ready to destack a plate to the Rotary Table, the arms, holding a plate, may extend with a plate only if there is not a plate in the corresponding position of the Rotary Table, and
- when a Stacker is ready to stack a plate from the Rotary Table, the arms may extend only if there is a plate in the corresponding position of the Rotary Table.

3. Creating an Abstract Model

The starting point of a development is an informal description of the system and its services incorporating safety properties. We want to create a specification of the system with a formal method in which we can perform consistency proofs and verify the safety properties. For a

smoother integration of formal methods also in industrial environments we need a graphical interface to the formal method. In order to achieve this we first model the informal requirements in UML. The UML diagrams are then translated to formal B-action systems with the tool U2B. The consistency proofs of the B-action systems can then be performed with the Atelier B tool.

3.1 UML-development incorporating safety aspects

UML (the Unified Modeling Language) is a graphical language for specifying, visualising, developing and documenting software-intensive systems [UML1.4]. The informal requirements of the system are first depicted with UML diagrams.

Use Cases and Component Diagrams. The functional requirements are captured together with their relationships in a use case diagram. Each use case expresses a service that the system will provide to a user. The reliability and safety issues of the system are given in the specification of the use cases as structured English text. The logically related use cases are determined, and grouped together into control system components in component and class diagrams. The component diagram is deduced from the use case diagram. Each use case can be mapped to a component service.

The Case Study. The functional requirements of the Stackers are to destack plates to the Rotary Table and to stack plates from the Rotary Table. A use case diagram of a Stacker is given in Figure 3.

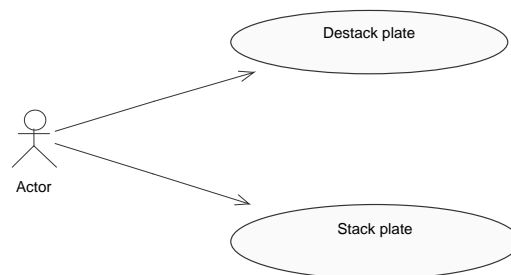


Figure 3. Use case diagram for the Stackers.

From the use case diagram in Figure 3 the component diagram in Figure 4 can be constructed. The Stackers interact with the Rotary Table by the services stacking and destacking plates. The Stackers are the active partners while the Rotary Table is the passive partner during the interaction. The Rotary Table in turn provides the service rotate which can be derived from the use case diagram of the Rotary Table (not shown).

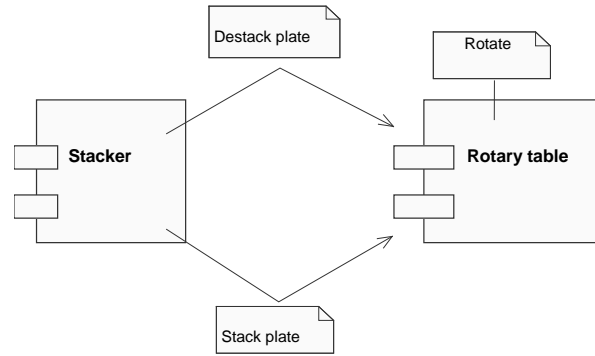


Figure 4. A component diagram for a Stacker and the Rotary Table.

Class- and Statechart Diagrams. The conceptual model of the component is given as active classes, *i.e.*, classes whose instances have autonomous behaviour. For each class we give the attributes and their types as well as the methods. The methods represent events that can occur spontaneously. However, we can also model methods that are callable by other methods in the component or from another component. Hence, the classes model the static behaviour of the component. In the most abstract class diagram we merely consider the attributes modelling the state and the command of the component. The methods consist of the main functionality of the system and the abstract representation of the classes of errors as well as the possibility to fix these errors.

The dynamic behaviour of the component is then specified with statechart diagrams. The services in the diagram are derived from the informal description of the services in the use case diagrams. We merely model state transitions and events causing these transitions at this level. In the absence of faults we go from state *Idle* to a service *n* when service *n* is requested. After the service is performed we return to state *Idle*. Already in the initial specification we consider the possibility of fault occurrence and system failure such as failures of the execution of command *service n* or a spontaneous fault occurrence even when a service is not requested. In all these failure transitions the system reacts to fault occurrence by entering state *Suspension*. From that state the system tries to execute a recovery procedure and continue functioning. When the fault tolerance limit has been reached and the system cannot carry out its functions anymore we have a failure of the system and enter state *Abort* which models a fail-safe state of the system.

The Case Study. The class diagram of the Stackers is given in Figure 5. In the specification the variable *scmd* modelling the commands given to the Stacker, as well as the variable *state* modelling the state of the Stacker are of interest. The variable *state* is not declared in the class. The tool U2B generates it automatically by checking the attached statechart diagrams. The multiplicity of the class is four, since there are four instances of Stacker. The operations capture the correct behaviour of the system as well as abstract representation of failures and their mitigation.



Figure 5. Specification class diagram for the Stacker unit.

The abstract statechart diagram of the Stacker for the service *stack* is shown in Figure 6. The states *idle*, *prepare*, *stack*, *prep_sk_suspended*, *st_suspended*, *suspended* and *abort*, form the states in this abstract statechart diagram. An instance of the Stacker component evolves from state *idle* to state *prepare* upon command *stack*, while the command, *scmd*, is set to *sk_cmd*. From the state *prepare* the Stacker evolves to the state *stack* performing the required service, provided that the command is *stack*, *sk_cmd*. In case the command is *destack*, *dsd_cmd*, it goes from state *prepare* to *destack*. When there is a failure in the execution of the command, the Stacker evolves to the state *prep_sk_suspended*. If a recovery procedure is found,

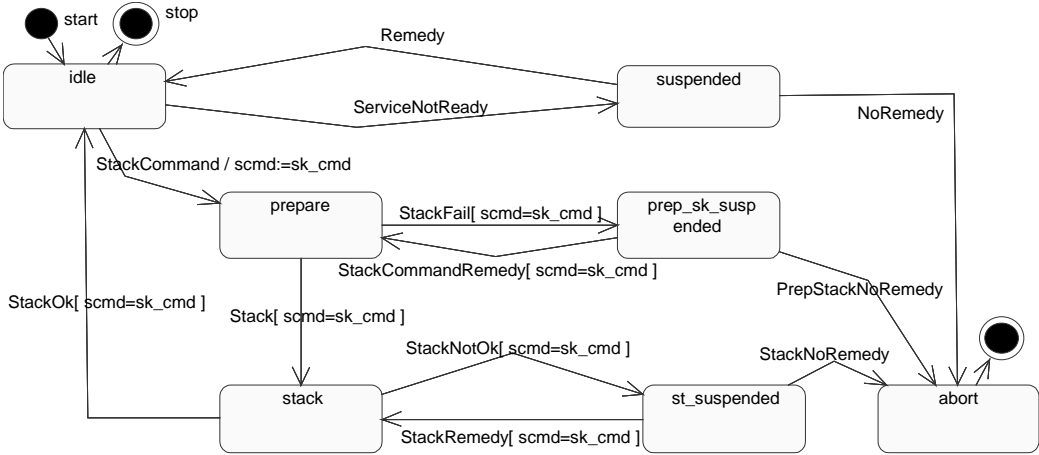


Figure 6. Specification statechart Diagram of a Stacker for the *stack* service.

the component can resume and go back to the state *prepare*. If the stack service is completed successfully, the component returns to its state *idle*. A failure in the execution of this service suspends the behaviour of the component, which evolves to the state *st_suspended*. If a remedy can be found, the object can resume its service providing state. If not, the exception is a failure and the Stacker enters the state *abort*. A corresponding diagram can be given for the service *destack* [Tsiopoulos03]. If a spontaneous fault occurs even when a service is not requested, the Stacker evolves from the state *idle* to the state *suspended*, wherefrom, if a remedy can be found, the component resumes to state *idle*. If there is no remedy, the Stacker aborts its execution.

3.2 Creating a B Model from UML

In order to ensure that the safety requirements of the system are satisfied in the initial specification, we need a formal analysis tool. A formal method that comes with such tools is the B Method [Abrial96]. We rely here on one of the tools supporting it, Atelier B [Steria96], when performing the development and the proving. In order to be able to reason about distributed systems within the B Method we use B-action systems [WS98] based on the action systems formalism [BS96] and related to Event based B [ClearSy01]. Event B also supports the development of distributed systems and indeed the extensions of event B are built on action systems. However, some features for action systems like procedure handling are not supported in event B yet. Procedures are important in distributed systems, since they provide more structuring to the systems, as well as a general communication mechanism between interacting systems.

The abstract specification. The first step in our formal development is to create an abstract B-action system from the abstract class- and statechart diagrams. The tool U2B [SB00,SnWa02] supports this translation. The B-action system is identified by a unique name. The attributes/variables of the system are given in the **VARIABLES**-clause. The types and the invariant properties of the local variables are given in the **INVARIANT**-clause and their initial value in the **INITIALISATION**-clause. The operations/services on the variables are given in the **OPERATIONS**-clause. The event triggering state transitions (names assigned to arrows) correspond to operation names in the abstract machine specification.

With the B-action system we model parallel and distributed systems, where operations are selected for execution in a non-deterministic manner. The operations are given in the form **Oper = SELECT P THEN S END**, where *P* is a predicate on the variables (also called a guard) and *S* is a substitution statement. When *P* holds the operation *Oper* is said to be enabled. Only enabled operations are considered for execution. When there are no enabled operations the system terminates. The operations are considered to be atomic, and hence, only their input-output behaviour is of interest.

Previously developed machines that have been shown correct can be used in the development by including them in the component using **SEES**, **INCLUDES** or **EXTENDS** [Abrial96]. The procedures declared in a B-action system can be local or global. The local procedures are declared and referenced locally within the same system. The global procedures, on the other

hand, may be referenced by other B-action systems as well. The global procedures are services provided/claimed between systems (components) and can be derived from the component diagram.

The U2B translator. A separate machine is created for each class and includes modelling of instances depending on the cardinality of the class. Attributes and (unidirectional) associations are translated into variables whose type is a function from the current instances to the attribute type (as defined in the Class diagram) or the associated class instances. Attribute types may be any valid B expression that defines a set. A simple example of a class with an association and an attribute and its corresponding B translation is shown in Figure 7. (A separate machine will be generated for class *B*).

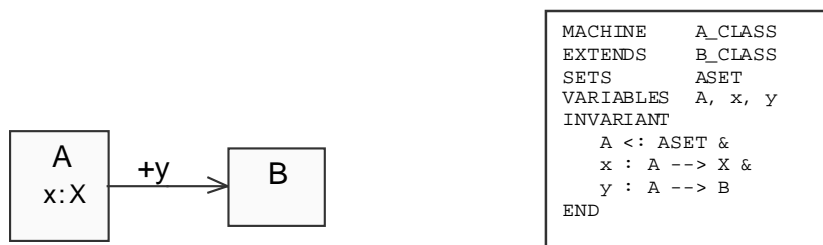


Figure 7. A simple class diagram and associated B machine.

Class cardinality may be fixed (set to a particular value), mutable (set to a range such as 0..n) or singleton (set to 1). For variable instance classes, instance modelling is by a variable and a create operation is automatically provided. For fixed instance classes, instances are modelled by a constant set of integers that are used to index the instances. If class cardinality is singleton, the U2B translator creates a machine without instance modelling.

In UML, model elements may have associated textual specification fields. For classes, one of these is a ‘documentation’ field. Any valid B clause can be added in the documentation field. For example, we use this method to specify invariants. Each clause must be headed by its B clause name.

The *dynamic behaviour* expressed in a class diagram is embodied in the behaviour specification of class operations and in class invariants. These details are specified either in a text field of the model element or in a statechart attached to the class. An operation may be specified completely by (pre-condition and semantics) text fields, completely by statechart transitions, or by a simultaneous composition of both.

In UML, text fields are provided for operations (as well as many other elements) and these are used to specify pre-conditions (for action systems these are interpreted as guards) and semantics for the operation. As there is no specific text field for a class invariant we use a clause in the documentation text box of the class' specification. UML does not impose a particular notation for these operation and invariant constraints. However, since we wish to translate to a B

machine we use B notation but support the object-oriented modelling conventions of implicit self-referencing and use of the dot notation for explicit instance references. Where a feature of a class is referenced without any instance being given, U2B adds a 'thisInstance' parameter and when a specific instance reference is given.

Invariants are generally of two kinds, instance invariants (describing properties that hold between the attributes and relationships within a single instance) and class invariants (describing properties that hold between different instances). For instance invariants U2B will add universal quantification over all instances of the class. For class invariants, the quantification over instances is an integral part of the property and must be given explicitly.

In UML a statechart model can be attached to a class to describe its behaviour. The U2B translator then combines the behaviour that the statechart model describes with any textual semantics of the operations. The name of the statechart model is used to define a state variable. The collection of states in the statechart model is used to define an enumerated set that is used in the type invariant of the state variable. The state variable is equivalent to an attribute of the class and may be referenced elsewhere in the class and by other classes. Statechart transitions define which operation call causes the state variable to change from the source state to the target state. To associate a transition with an operation, the transition's event must be the name of the operation. Additional guard conditions can be attached to a transition to constrain when it can take place. Transitions cause the implicit action of changing the state variable from the source state to the target state. Additional actions can also be attached to transitions. The translator finds all transitions associated with an operation and compiles a **SELECT**-substitution of the following form

```
SELECT statevar=sourcestate1 & sourcestate1_guards
THEN statevar:=targetstate1 || targetstate1_actions
WHEN statevar=sourcestate2 & sourcestate2_guards
THEN statevar:=targetstate2 || targetstate2_actions
ELSE skip END ||
<operation body from semantics window>
```

The Case Study. The abstract class and statechart diagrams of the Stackers can be translated to B-action systems, using the U2B tool, in order to be able to verify them with the Atelier B tool. U2B checks the multiplicity of the class, in order to create instances of the class if it is required. The multiplicity of the Stacker is four. Hence, the tool generates a constant STACKER of type 1..4 in the STACKER_CLASS. Furthermore, U2B checks the attached statechart diagrams of the class and translates automatically the set of states and the state variable, for which the name is the name of the state machine in UML. The translated specification machine for the stack operation of the Stacker can be found below.

```

MACHINE    STACKER_CLASS_STACKER_R1
SEES
    def
SETS      /* Stacker states */
    S_STATE = {idle, prepare, destack, des_suspended, prep_dsk_suspended,
               stack, prep_sk_suspended, st_suspended, suspended, abort}
CONSTANTS
    STACKER
PROPERTIES
    STACKER = 1..4 /* Four Stacker instances */
VARIABLES
    s_state, /* State */
    scmd     /* Command */
INVARIANT
    s_state : STACKER --> S_STATE &
    scmd : STACKER --> SCOMMAND &
    !(thisSTACKER).(thisSTACKER:STACKER & s_state(thisSTACKER) = destack =>
                    scmd(thisSTACKER) = dsk_cmd) &
    !(thisSTACKER).(thisSTACKER:STACKER & s_state(thisSTACKER) = stack =>
                    scmd(thisSTACKER) = sk_cmd)
INITIALISATION
    s_state:= %xx.(xx:STACKER | idle) ||
    ANY yy WHERE yy:SCOMMAND THEN scmd:= %xx.(xx:STACKER | yy) END
OPERATIONS
    /* Operation for supplying the stack command, "sk_cmd". */
    StackCommand (thisSTACKER) =
    PRE thisSTACKER : STACKER THEN
        SELECT s_state(thisSTACKER)=idle
        THEN s_state(thisSTACKER):=prepare || scmd(thisSTACKER):=sk_cmd END
    END;
    ...
    /* Stack a plate */
    Stack (thisSTACKER) =
    PRE thisSTACKER : STACKER THEN
        SELECT s_state(thisSTACKER)=prepare & scmd(thisSTACKER)=sk_cmd
        THEN s_state(thisSTACKER):=stack END
    END;
    /* The Stacker has stacked a plate */
    StackOk (thisSTACKER) =
    PRE thisSTACKER : STACKER THEN
        SELECT s_state(thisSTACKER)=stack & scmd(thisSTACKER)=sk_cmd
        THEN s_state(thisSTACKER):=idle END
    END;
    /* Failure - The Stacker could not stack a plate */
    StackNotOk (thisSTACKER) =
    PRE thisSTACKER : STACKER THEN
        SELECT s_state(thisSTACKER)=stack & scmd(thisSTACKER)=sk_cmd
        THEN s_state(thisSTACKER):=st_suspended END
    END;

```

```

        /* Fix the problem */
StackRemedy (thisSTACKER) =
PRE thisSTACKER : STACKER THEN
    SELECT s_state(thisSTACKER)=st_suspended & scmd(thisSTACKER)=sk_cmd
    THEN s_state(thisSTACKER):=stack END
END;

        /* Abort - The problem cannot be fixed */
StackNoRemedy (thisSTACKER) =
PRE thisSTACKER : STACKER THEN
    SELECT s_state(thisSTACKER)=st_suspended THEN s_state(thisSTACKER):=abort END
END;

...

        /* General "not ready" failure */
ServiceNotReady (thisSTACKER) =
PRE thisSTACKER : STACKER THEN
    SELECT s_state(thisSTACKER)=idle THEN s_state(thisSTACKER):=suspended END
END;

        /* Fix the problem */
Remedy (thisSTACKER) =
PRE thisSTACKER : STACKER THEN
    SELECT s_state(thisSTACKER)=suspended THEN s_state(thisSTACKER):=idle END
END;

        /* No remedy - Abort */
NoRemedy (thisSTACKER) =
PRE thisSTACKER : STACKER THEN
    SELECT s_state(thisSTACKER)=suspended THEN s_state(thisSTACKER):=abort END
END
END

```

Figure 8 shows the Rational Rose window of the Stacker class diagram. The **SEES** and **INVARIANT** clauses are given in the documentation box of the class specification window. The U2B-tool appends this invariant to the attributes and their types given in the class diagram when creating the **INVARIANT** clause of the abstract B specification. For the Stackers the attribute *scmd*, which can be assigned values from the set SCOMMAND, consisting of the commands *stack*, *sk_cmd*, and *destack*, *dsk_cmd*, is of type:

scmd : STACKER → SCOMMAND.

Hence, every instance of the Stacker has a command variable, *scmd*, which can be assigned the value *sk_cmd* or *dsk_cmd*.

The state variable is automatically initialized by U2B, which checks the starting point in the statechart diagrams. The start point in the diagrams of the Stackers is the state *idle*. The initial value of a variable in the class diagram is either explicitly given in the diagram or assumed to be any value of the type set of the variable. The initial value of the variable *scmd* is not explicitly given. Hence, each Stacker is initialised to have either the command *stack* or *destack*.

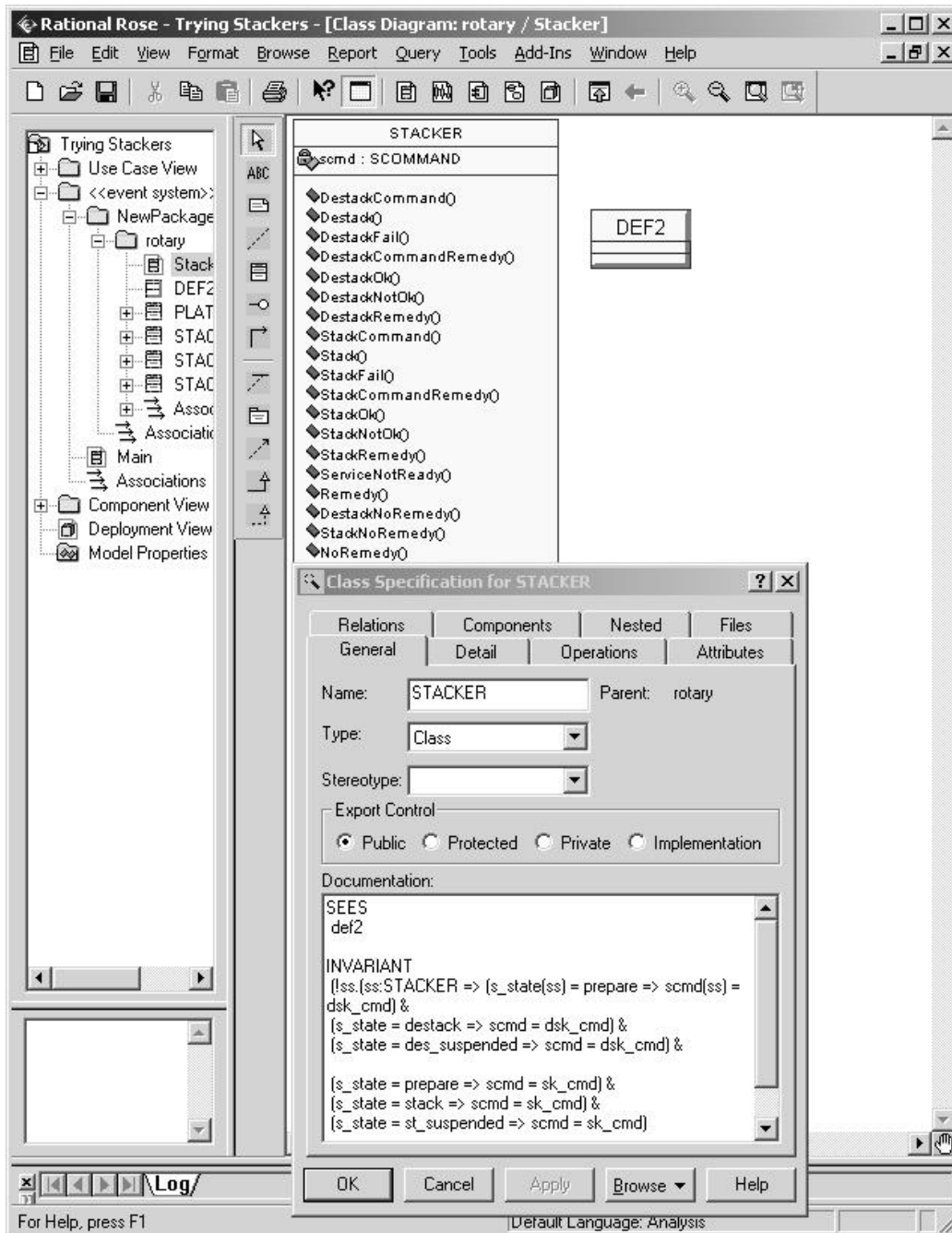


Figure 8. Class diagram of the Stacker.

The operation names in the class diagram in Figure 5 correspond to the names on the state transitions in the statechart diagram in Figure 6. If an operation name does not have a transition the body of this operation is considered to be *skip*. The operations in the class- and statechart diagrams are translated to operations in the abstract machine specification. The transition *StackCommand* in Figure 6 is translated to the following abstract machine operation:

```

StackCommand(thisSTACKER) =
    PRE thisSTACKER : STACKER THEN
        SELECT s_state(thisSTACKER) = idle
        THEN s_state(thisSTACKER) := prepare || scmd(thisSTACKER) := sk_cmd END
    END

```

Since we have several instances of a Stacker, each operation has a parameter `thisSTACKER` of type `STACKER`. The operation *StackCommand* is enabled when the Stacker instance `thisSTACKER` is idle. The Stacker instance evolves to the state *prepare* and the command is set to *stack*. The tool Atelier B generated 150 proof obligations for the machine specification. The automatic prover could prove 80 of these proof obligations. The rest were discharged with the interactive prover.

4. Refining the system

An important feature provided by the B-action systems formalism is the possibility to stepwise refine specifications. The refinement is a process transforming a specification *A* into a system *C* when *A* is abstract and non-deterministic and *C* is more concrete, more deterministic and preserves the functionality of *A*. We use a particular refinement method, where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behaviour. This type of refinement is referred to as *superposition refinement* [BaKu83, Katz93]. When dealing with complex control systems it is especially convenient to stepwise introduce details about the system to the specification and not to have to handle all the implementation issues at once. In the refinement process we identify the attributes suggested in the requirements specification. These attributes/variables are then added gradually to the specification with their safety conditions and properties. We add the computation concerning the new variable to the existing operations by strengthening their guards and adding new substitutions on the variables. New operations that only assign the new variables may also be introduced.

As the system development proceeds we obtain more elaborated information about faults and conditions of failure occurrence [PTWBEJ01, Troubitsyna00]. The refinement step introduces a distinction between faults. We also introduce a distinction between different repair procedures by adjusting the *Remedy* operation for each fault accordingly.

When all the required features have been added to the components of the system, each component is decomposed into control systems modules, a *plant*, a *controller*, *sensors* and *actuators* [PRWJ01, Sekerinski98]. The plant specification describes autonomous behaviour of the component, whereas the controller describes algorithms that guide the plant behaviour. Thus, the role of the controller is to react to changes in the plant. The sensors convert measurements from the plant into readings for the controller. Correspondingly, the actuators convert commands from the controller into control signals to the plant.

4.1 Refinement with UML and B

In order to get a graphical interface to the formal method to facilitate the reading and analysing of the development, UML artefacts are integrated in the formal development process. The stepwise introduction of implementation details (features) can be applied by adding these features into class and statechart diagrams. Part of the refined diagrams is automatically generated from the more abstract ones. The new features are modelled with new attributes. Sometimes even new operations have to be added in the class diagrams. The more concrete behaviour of the system can then be modelled with new states and more complex transitions in the statechart diagrams taking into consideration the new variables and operations of the corresponding class diagram. While adding the new features to the statecharts, the refinement rules for B-action systems should be applied [WS98]. For each refinement step a new set of class and statechart diagrams are generated. The refined diagrams are then automatically translated to B-action systems with the U2B-tool.

When refining the statechart diagrams a single transition, representing one event can be broken down into several different transitions representing alternative events that are either refinements of the original event or additional events (refining the non-event, skip). Moreover, a single state can be broken down into several sub-states. Currently, the U2B translation does not support hierarchical states. Therefore, we refine the original set of states with a complete new set of states replacing the refined state with its sub-states. This involves renaming all the states, even those that are not being decomposed.

The correspondence of all the states has to be explicitly stated in the invariant even when there is no real change to the state. An improvement to the U2B translator, so that it handles hierarchical states would enable the implicit refinement correspondence of B to be utilised. The hierarchical correspondence would become more obvious and the proofs easier to discharge. Currently the U2B tool provides the following support for refinement.

- a) The B component is created as a refinement instead of a machine (*i.e.*, it has a **REFINEMENT** header and a **REFINES**-clause).
- b) Refinement of events can be indicated on statecharts by merging and splitting transitions on statecharts.

The Case Study. The class diagram of the refinement of the Stacker unit can be found in Figure 9. A more complete diagram can be found elsewhere[Tsiopoulos03]. In this refinement step we introduce the features concerning the arms, positions, as well as the capacity of each Stacker.

A variable *arms* is introduced modelling the position, *up* and *down*, of the arms of a Stacker. Moreover, a variable *s_pos* is added to model the position (positions 2, 3, 5, or 6) of every stacker above the Rotary Table. Since the Rotary Table needs to know the values of these variables, they are declared and assigned in the local procedures machine STACKER_PROC. A variable *full* is introduced modeling a Stacker sensing whether it is full or not. If a Stacker is full, it cannot stack any more plates from the Rotary Table. The variable *empty* is added to model a Stacker sensing whether it is empty or not. If a Stacker is empty, it cannot destack any

plate to the Rotary Table. The variables *full* and *empty* are local to the Stackers and, hence, given in the machine STACKER_R1.

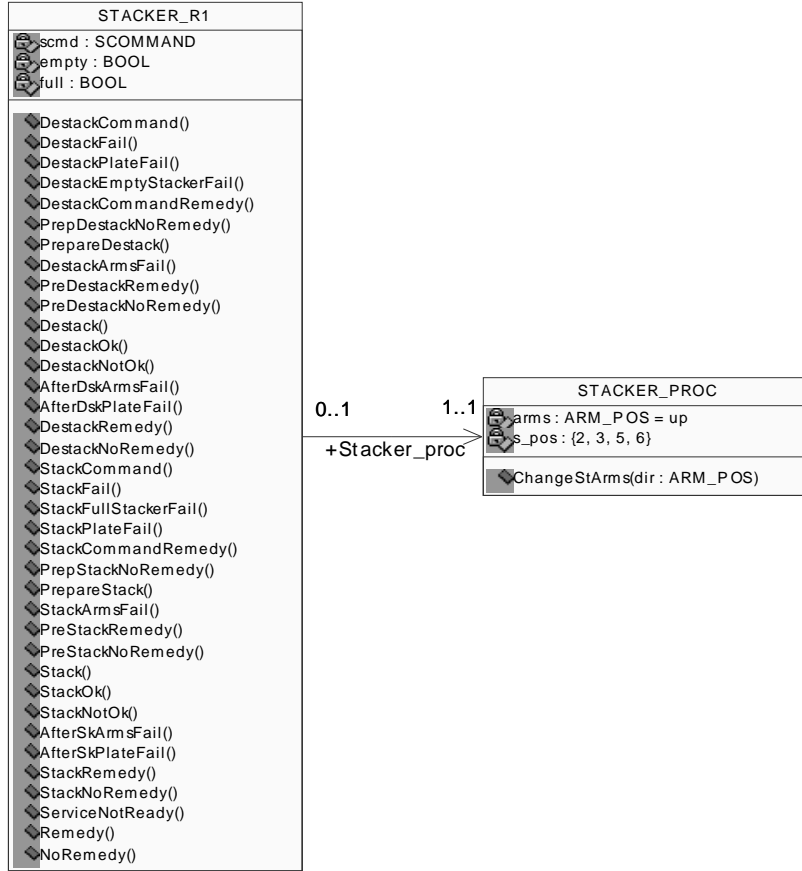


Figure 9. The class diagram of the refined Stacker.

The invariant of the refinement machine STACKER_R1 gives the relation between the variables of this refined machine and the variables of the machine specification STACKER that it refines. A part of the invariant for the Stacker is given below. A more detailed invariant can be found in Appendix A and in [Tsiopoulos03].

$$\begin{aligned}
 &\forall \text{thisSTACKER} . (\text{thisSTACKER} \in \text{STACKER} \wedge \\
 &\quad \text{s_state1}(\text{thisSTACKER}) = \text{stack1} \Rightarrow \text{scmd}(\text{thisSTACKER}) = \text{sk_cmd} \wedge \\
 &\quad \text{s_state1}(\text{thisSTACKER}) = \text{destack1} \Rightarrow \text{scmd}(\text{thisSTACKER}) = \text{dsk_cmd} \wedge \\
 &\quad \text{s_state1}(\text{thisSTACKER}) = \text{ready_to_destack1} \Rightarrow \text{arms}(\text{Stacker_proc}(\text{thisSTACKER})) = \text{down} \wedge \\
 &\quad \text{s_state1}(\text{thisSTACKER}) = \text{ready_to_stack1} \Rightarrow \text{arms}(\text{Stacker_proc}(\text{thisSTACKER})) = \text{down})
 \end{aligned}$$

The invariant states that when a Stacker is in the state *ready_to_destack1*, the *arms* of that Stacker have to be down. Furthermore, when a Stacker is in the state *stack1* or *destack1*, the command, *scmd*, has to be equal to the command *stack*, *sk_cmd*, or *destack*, *dsk_cmd*, respectively.

The stack operation of the refined Stacker is shown in Figure 10. Decision pseudo states have been added in this diagram to reduce guard complexity. There is a corresponding statechart diagram for the service *destack* of Stacker (not shown). A new state *ready_to_stack1* is added between the states *prepare1* and *stack1*. From state *prepare1* the Stacker evolves to the state *ready_to_stack1* provided the command is equal to stack, *sk_cmd*, the Stacker is not full, and the plate holder under this stacker holds a plate. The Stacker *arms* are moved to position *down*, in order for the Stacker to be able to grip the plate. The transitions going to the new states *ready_to_stack1* and *prestack_suspended1* are added in this refinement step.

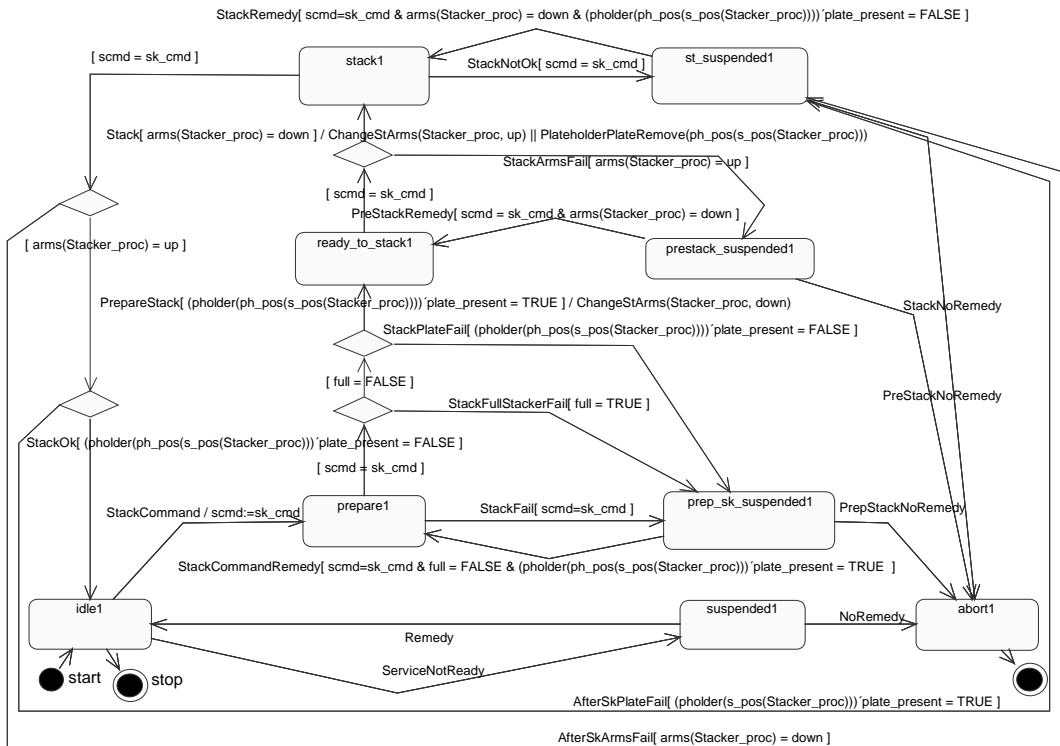


Figure 10. The refined statechart diagram for service stack.

If an exception occurs the component evolves from the state *prepare1* to the state *prep_sk_suspended1*. Additionally to the transition *StackFail* two new failure transitions have been added. The transition *StackFullStackerFail* suspends the Stacker, if it is *full*. The transition *StackPlateFail* is enabled, if there is not a plate present on the plate holder under this Stacker. The suspended Stacker can resume and continue from the state *prepare1* if a remedy can be found.

From the state *ready_to_stack1* the component evolves to the state *stack1* provided that the command is stack, *sk_cmd*, and the *arms* are down ready to take up the plate. The action moves up the arms and removes the plate from the plateholder by calling appropriate procedures. If the *arms* were not *down*, the Stacker becomes suspended. The component evolves back to the state

idle1 from state *stack1*, provided that the *arms* of this Stacker are *up* and there is not a plate present under this Stacker. If the *arms* are *down* or there is a plate on the plate holder, the component is suspended.

The service *destack* is similar to *stack* above. The command should be *destack*, *dsk_cmd*, the Stacker must not be *empty*, and there should be no plate on the plate holder under this Stacker. If this holds the arms are moved up and a plate is added to the plate holder on the Rotary Table under this Stacker.

4.2 Proofs in B

Using AtelierB we can formally prove that the refinement is sound. For this a number of proof obligations [BW98, WS98] are generated automatically by Atelier B with the help of the Evt2b translator [ClearSy01]. These proof obligations can be discharged using the autoprover and the interprover in AtelierB. The proof obligations that are generated for a refinement check that the initialisation of the refinement establishes the invariant and that each operation and global procedure in the refinement preserves the invariant. Furthermore, it is checked that the auxiliary operations should only change the variables that are added in the refinement step. Using the Evt2b translator proof obligations on the non-divergence of the auxiliary operations are generated. With the help of the Evt2b translator it is also checked that the more abstract system terminates, if the refined system does. Moreover, if a global procedure is enabled in the more abstract system, it should also be enabled in the refined system or a finite sequence of actions should result in its guard becoming true. If the occurrence of a certain fault is considered in the abstract system, it should also be considered in the more concrete system.

The Case Study. The class and statechart diagrams of the refined Stackers are translated to B-action systems using the U2B tool. In the class diagram the refinement is modelled with a realize relationship. U2B also interprets class dependency relationships with stereotype bind as class instantiation and adds an **INCLUDES** clause with the names of the instantiated class in the STACKER_R1 refinement machine. (An **INCLUDES** clause can also be given manually in the documentation box of the classes).

The STACKER_PROC class containing the procedures does not have any statechart diagram attached. For each procedure/operation there is a specification window for the *parameters*, *preconditions*, and *semantics*. When the logical view package in the UML model has the stereotype *action system*, the U2B tool translates the conditions in the precondition box of an operation to a guard instead of a pre-condition.

A feature in the Stacker's first refinement class diagram in Figure 9 is the *association* UML relationship between the STACKER_R1 class and the procedure class of the Stacker. The procedure class STACKER_PROC has the same multiplicity, four, as the STACKER_R1 class. The instances are related one to one as a *total injection* in the class STACKER_R1.

Stacker_proc : STACKER >→ STACKER_PROC

Whenever a procedure is called from the `STACKER_PROC` class, the `Stacker_proc` variable is used as a parameter in order for U2B to automatically generate the instance parameter `Stacker_proc(thisSTACKER)`. Moreover, the variables of the machine `STACKER_PROC` are given as `arms(Stacker_proc)` and `s_pos(Stacker_proc)` in the statechart diagram in Figure 10 to generate the correct instances in the refinement machine `STACKER_R1` in Appendix A. For this refinement step 1050 proof obligations were generated. Of these proof obligations 151 could be discharged automatically and the rest were easily proven with the interactive prover of Atelier B.

In the next refinement step the number of plates in each Stacker is explicitly expressed. The grippers of the arms are also modelled as a feature in this step. As a final refinement step the Stackers will be decomposed into plant, controller, sensors and actuators. However, these steps are not shown in this paper.

5. Conclusions

In this paper we have developed part of a safety critical healthcare system applying a methodology integrating UML, action systems, B and safety analysis. We have used UML as a graphical interface to the formal methods. We model the requirements in UML and stepwise add more features to the UML diagrams to get a model of the final system. Throughout the development we capture the safety properties of the system in the model. In order to ensure that these safety properties are satisfied in the model, we need to formally prove the model. This can be done within the B-action system formalism using the provers of a B tool such as Atelier B. The B-action systems formalism is an extension of the B Method supporting the development of distributed systems. For the translation of the UML model to B-action systems we use the tool U2B. This tool automatically translates class and statechart diagrams to B machines. This automatic translation plays an important role in the development process.

The combination of UML and B-action systems in the development process has proved useful in industrial strength projects [PTWBEJ01] where tightened regulatory requirements make the use of formal methods necessary. Using UML as a graphical interface to the formal method makes it easier to understand the formal development and facilitates the integration of formal methods into the current industrial development process.

The UML and B translations used were initially based on the translation of OMT class diagrams into B specifications as presented by Meyer et al [MeSo99]. Our translation has, though, a greater concern for the provability and refinement consistency of the generated B specification. Sekerinski et al [SeZu02] translates UML statecharts to B supporting hierarchy, concurrency and communication. However, they only want to provide a formalisation of statecharts and to find a way to formally develop statecharts. Furthermore, we intend to develop the U2B tool to better support superposition refinement of state charts by supporting hierarchical states and providing assistance in generating the corresponding refinement invariant.

References

- [Abrial96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [BaKu83] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proc. of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.
- [BS96] R.J.R. Back and K. Sere. From modular systems to action systems. *Software -- Concepts and Tools 17*, pp. 26-39, 1996.
- [BJW03] P. Boström, M. Jansson, and M. Walden. A healthcare case study: Fillwell. TUCS Technical Reports, Turku Centre for Computer Science, Finland. To appear.
- [BW98] M. Butler and M. Walden. Parallel programming with the B Method. Chapter 5 in [SS98], pp 183-195.
- [ClearSy01] *Event B Reference Manual v1*. ClearSy, 2001.
- [Katz93] S.M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337-356, April 1993.
- [MeSo99] Meyer, E. & Souquieres, J. A Systematic approach to Transform OMT Diagrams to a B specification. *Proceedings of the FM'99*, LNCS 1708 1, pp. 875-895, 1999.
- [PE01] Fillwell™2002 – Features Guide. <http://www.abo.fi/~marina.walden/fillwell.pdf>.
- [PRWJ01] L. Petre, M. Rönkkö, M. Waldén, and M. Jansson. *Methodology of co-design based on the healthcare case study*. TUCS Technical Reports No 437, Turku Centre for Computer Science, Finland. October 2001.
- [PS00] L. Petre and K. Sere. Developing Control Systems Components. In *Proceedings of IFM'2000 - Second International Conference on Integrated Formal Methods*, Germany, November 2000. LNCS 1945, pp. 156-175, Springer-Verlag.
- [PTW02] L. Petre, E. Troubitsyna and M.Walden, A Healthcare Case Study. In *Proceedings of RCS'02 - International workshop on Refinement of Critical Systems: Methods, Tools and Experience*, Grenoble, France, January 2002.
- [PTWBEJ01] L. Petre, E. Troubitsyna, M. Waldén, P. Boström, N. Engblom, and M. Jansson. *Methodology of integration of formal methods within the healthcare case study*. TUCS Technical Reports No 436, Turku Centre for Computer Science, Finland. October 2001.
- [Sekerinski98] E. Sekerinski. Production cell. Chapter 6 in [SS98], pp. 197-254.
- [SS98] E. Sekerinski and K. Sere (eds.). *Program Development by Refinement - Case Studies Using the B Method*. Springer-Verlag, 1998.

- [SeZu02] E. Sekerinski and R. Zurob. Translating Statecharts to B. *Proceedings of Integrated Formal Methods (IFM'02)*, pp. 128 - 144. Springer, 2002.
- [SB00] C. Snook and M. Butler. U2B Downloads.
<http://www.ecs.soton.ac.uk/~cfs/U2Bdownloads.htm>
- [SnWa02] C. Snook and M. Walden, Use of U2B for Specifying B Action Systems. In *Proceedings of RCS'02 - International workshop on Refinement of Critical Systems: Methods, Tools and Experience*, Grenoble, France, January 2002.
- [Steria96] Stéria Méditerranée. *Atelier B*. France, 1996.
- [Troubitsyna00] E. Troubitsyna. *Stepwise Development of Dependable Systems*. Turku Centre for Computer Science, TUCS, Ph.D. thesis No.29. June 2000.
- [Tsiopoulos03] L. Tsiopoulos. "UML modelling of control systems". Master Thesis, Department of Computer Science, Aabo Akademi University, Finland, March 2003. http://www.abo.fi/~ltsiopou/Master_Thesis.pdf.
- [UML1.4] Unified Modeling Language (UML) 1.4 specification,
<http://cgi.omg.org/cgi-bin/doc?formal/01-09-67>
- [WS98] M. Walden and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design* 13(5-35), 1998. Kluwer Academic Publishers.

Appendix A: The first refinement step of the Stackers.

Part of the refinement machine STACKER_R1_CLASS and its local procedures in machine STACKER_PROC_CLASS.

```
REFINEMENT STACKER_R1_CLASS
REFINES
  STACKER_CLASS_STACKER_R1
SEES
  def2
INCLUDES
  PLATEHOLDER
EXTENDS
  STACKER_PROC_CLASS
SETS
  S_STATE1={idle1,destack1,stack1,des_suspended1,st_suspended1,prepare1, ready_to_destack1,
            ready_to_stack1, prep_sk_suspended1, prestack_suspended1, prep_dsk_suspended1,
            predestack_suspended1, suspended1, abort1}
VARIABLES
  s_state1, scmd, empty, full, Stacker_proc
INVARIANT
  s_state1 : STACKER --> S_STATE1 &
  scmd : STACKER --> SCOMMAND &
  empty : STACKER --> BOOL &
  full : STACKER --> BOOL &
  Stacker_proc : STACKER >> STACKER_PROC &

  !(thisSTACKER).(thisSTACKER:STACKER & s_state(thisSTACKER) = idle =>
    s_state1(thisSTACKER) = idle1) &
  !(thisSTACKER).(thisSTACKER:STACKER & s_state1(thisSTACKER) = idle1 =>
    s_state(thisSTACKER) = idle) &

  !(thisSTACKER).(thisSTACKER:STACKER & s_state(thisSTACKER) = prepare =>
    s_state1(thisSTACKER) = prepare1) &
  !(thisSTACKER).(thisSTACKER:STACKER & s_state1(thisSTACKER) = prepare1 =>
    s_state(thisSTACKER) = prepare ) &
  !(thisSTACKER).(thisSTACKER:STACKER & s_state1(thisSTACKER) = ready_to_destack1 =>
    s_state(thisSTACKER) = prepare ) &
  !(thisSTACKER).(thisSTACKER:STACKER & s_state1(thisSTACKER) = ready_to_stack1 =>
    s_state(thisSTACKER) = prepare) &
  ...
  /* When Stacker is in the states ready_to_destack1 and ready_to_stack1, the arms have to be down */
  !(thisSTACKER).(thisSTACKER:STACKER & s_state1(thisSTACKER) = ready_to_destack1 =>
    arms(Stacker_proc(thisSTACKER)) = down) &
  !(thisSTACKER).(thisSTACKER:STACKER & s_state1(thisSTACKER) = ready_to_stack1 =>
    arms(Stacker_proc(thisSTACKER)) = down) &
  ...
INITIALISATION
  s_state1:= %xx.(xx:STACKER | idle) ||
  ANY yy WHERE yy:SCOMMAND THEN scmd:= %xx.(xx:STACKER | yy) END ||
  ANY yy WHERE yy:BOOL THEN empty:= %xx.(xx:STACKER | yy) END ||
  ANY yy WHERE yy:BOOL THEN full:= %xx.(xx:STACKER | yy) END ||
  ANY yy WHERE STACKER_PROC THEN Stacker_proc:= %xx.(xx:STACKER | yy)
```

OPERATIONS

```
StackCommand (thisSTACKER) =          /* Operation for supplying the stack command, "sk_cmd".*/
PRE thisSTACKER : STACKER THEN
    SELECT s_state1(thisSTACKER)=idle1
    THEN s_state1(thisSTACKER):=prepare1 || scmd(thisSTACKER):=sk_cmd END
END;

StackFail (thisSTACKER) =              /* General failure – Suspend service */
PRE thisSTACKER : STACKER THEN
    SELECT s_state1(thisSTACKER)=prepare1 & scmd(thisSTACKER)=sk_cmd
    THEN s_state1(thisSTACKER):=prep_sk_suspended1 END
END;

StackFullStackerFail (thisSTACKER) =   /* The Stacker is full */
PRE thisSTACKER : STACKER THEN
    SELECT full(thisSTACKER) = TRUE
    THEN s_state1(thisSTACKER):=prep_sk_suspended1 ||
        SELECT s_state1(thisSTACKER)=prepare1 & scmd(thisSTACKER) = sk_cmd THEN skip END
    END
END;

StackPlateFail (thisSTACKER) =        /* There is no plate under the Stacker */
PRE thisSTACKER : STACKER THEN
    SELECT (pholder(ph_pos(s_pos(Stacker_proc(thisSTACKER)))))'plate_present = FALSE
    THEN s_state1(thisSTACKER):=prep_sk_suspended1 ||
        SELECT full(thisSTACKER) = FALSE THEN
            SELECT s_state1(thisSTACKER)=prepare1 & scmd(thisSTACKER) = sk_cmd
            THEN skip END
        END
    END
END;

...

PrepareStack (thisSTACKER) =          /* Prepare to stack with the arms down, the Stacker not full and */
PRE thisSTACKER : STACKER THEN /* a plate present under the Stacker */
    SELECT (pholder(ph_pos(s_pos(Stacker_proc(thisSTACKER)))))'plate_present = TRUE
    THEN s_state1(thisSTACKER):=ready_to_stack1 || ChangeStArms(Stacker_proc(thisSTACKER), down) ||
        SELECT full(thisSTACKER) = FALSE THEN
            SELECT s_state1(thisSTACKER)=prepare1 & scmd(thisSTACKER) = sk_cmd
            THEN skip END
        END
    END
END;

StackArmsFail (thisSTACKER) =         /* Suspend – The arms are still up */
PRE thisSTACKER : STACKER THEN
    SELECT arms(Stacker_proc(thisSTACKER)) = up
    THEN s_state1(thisSTACKER):=prestack_suspended1 ||
        SELECT s_state1(thisSTACKER)=ready_to_stack1 & scmd(thisSTACKER) = sk_cmd
        THEN skip END
    END
END;

PreStackRemedy (thisSTACKER) =       /* Fix the problem */
PRE thisSTACKER : STACKER THEN
    SELECT s_state1(thisSTACKER)=prestack_suspended1 &
        scmd(thisSTACKER) = sk_cmd & arms(Stacker_proc(thisSTACKER)) = down
    THEN s_state1(thisSTACKER):=ready_to_stack1 END
END;
```

```

PreStackNoRemedy (thisSTACKER) =          /* Abort – The problem cannot be fixed */
PRE thisSTACKER : STACKER THEN
    SELECT s_state1(thisSTACKER)=prestack_suspended1 THEN s_state1(thisSTACKER):=abort1 END
END;
Stack (thisSTACKER) =                      /* Stack plate */
PRE thisSTACKER : STACKER THEN
    SELECT arms(Stacker_proc(thisSTACKER)) = down
    THEN s_state1(thisSTACKER):=stack1 || ChangeStArms(Stacker_proc(thisSTACKER), up) ||
        /* Call procedure from the PLATEHOLDER class to remove the plate under this Stacker */
        PlateholderPlateRemove(ph_pos(s_pos(Stacker_proc(thisSTACKER)))) ||
    SELECT s_state1(thisSTACKER)=ready_to_stack1 & scmd(thisSTACKER) = sk_cmd
    THEN skip END
    END
END;
StackOk (thisSTACKER) =                   /* The Stacker has stacked the plate */
PRE thisSTACKER : STACKER THEN
    SELECT (pholder(ph_pos(s_pos(Stacker_proc(thisSTACKER))))).plate_present = FALSE
    THEN s_state1(thisSTACKER):=idle1 ||
        SELECT arms(Stacker_proc(thisSTACKER)) = up THEN
            SELECT s_state1(thisSTACKER)=stack1 & scmd(thisSTACKER) = sk_cmd
        THEN skip END
    END
    END
END;
StackNotOk (thisSTACKER) =               /* Failure – The Stacker has not stacked the plate */
PRE thisSTACKER : STACKER THEN
    SELECT s_state1(thisSTACKER)=stack1 & scmd(thisSTACKER) = sk_cmd
    THEN s_state1(thisSTACKER):=st_suspended1 END
END;
AfterSkArmsFail (thisSTACKER) =         /* Failure - After stack the arms are down */
PRE thisSTACKER : STACKER THEN
    SELECT arms(Stacker_proc(thisSTACKER)) = down
    THEN s_state1(thisSTACKER):=st_suspended1 ||
        SELECT s_state1(thisSTACKER)=stack1 & scmd(thisSTACKER) = sk_cmd THEN skip END
    END
END;
AfterSkPlateFail (thisSTACKER) =       /* Failure – There is still plate under the Stacker */
PRE thisSTACKER : STACKER THEN
    SELECT (pholder(s_pos(Stacker_proc(thisSTACKER))))).plate_present = TRUE
    THEN s_state1(thisSTACKER):=st_suspended1 ||
        SELECT arms(Stacker_proc(thisSTACKER)) = up THEN
            SELECT s_state1(thisSTACKER)=stack1 & scmd(thisSTACKER) = sk_cmd THEN skip END
        END
    END
END;
...
END

```

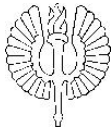
```

MACHINE STACKER_PROC_CLASS
SEES
    def2
CONSTANTS
    STACKER_PROC
PROPERTIES
    STACKER_PROC = 1..4
VARIABLES
    arms, s_pos
INVARIANT
    arms : STACKER_PROC --> ARM_POS &
    s_pos : STACKER_PROC --> {2, 3, 5, 6} /* Stacker position on the table */
INITIALISATION
    arms := %xx.(xx:STACKER_PROC | up) || /* The arms of all the Stackers are up in the beginning */
    s_pos := {1|->2, 2|->3, 3|->5, 4|->6} /* Stackers 1, 2, 3, and 4 are above the Rotary Table
                                           positions 2, 3, 5 and 6, respectively */
OPERATIONS
    /* Local procedure to change the position of the Stacker's arms */
    ChangeStArms (thisSTACKER_PROC,dir) =
    PRE thisSTACKER_PROC : STACKER_PROC & dir:ARM_POS THEN arms(thisSTACKER_PROC) := dir END
END

```

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

<http://www.tucs.fi/>



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research



Turku School of Economics and Business Administration

- Institute of Information Systems Science