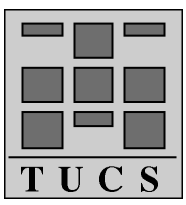
A Healthcare Case Study: Fillwell

Pontus Boström,
Micaela Jansson,
Marina Waldén

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



Turku Centre for Computer Science
TUCS Technical Report No 569
December 2003
ISBN 952-12-1260-8
ISSN 1239-1891

Abstract

We describe a case study on a liquid handling workstation, Fillwell, that has been conducted within the EU-project MATISSE as a co-operation between academia and industry. Since the workstation is a safety-critical system that need to operate with very high precision, it need to be safe and very reliable. These aspects are achieved by developing the system using formal methods where the safety analysis goes hand in hand with the formal development. We use the B Action Systems formalism for the development, where we can benefit from the properties of action systems for designing distributed systems and on the tool support via the B Method. The development is performed in a stepwise manner adding new features to the system in each step. We use UML as a graphical interface to the formal methods to achieve a better acceptance of the methodology by the industrial partner. UML diagrams are created for all the refinement steps. Hence, UML provides us with a documentation of the whole development process. The stepwise development and the graphical interface of our method has shown to be a suitable approach for applying formal methods on this industrial sized case study.

Keywords: formal methods, industrial application, B Method, Action Systems, UML, stepwise refinement

TUCS Laboratory
Distributed Systems Laboratory

1. Introduction

In this paper we describe the healthcare case study within the EU-project MATISSE[1] [MATISSE03]. This case study was conducted by Åbo Akademi University in co-operation with Wallac, a division of Perkin Elmer Lifesciences. It deals with the development of a safety-critical drug discovery system, Fillwell [PE01]. The Fillwell system is a control system that should guarantee an extremely high precision and a constant level of quality on the experiments to be performed. Even if the direct harm to the humans using the system is quite moderate, the indirect harm caused by the results of incorrectly performed experiments could be catastrophic. Hence, safety and reliability are important issues for this system. These aspects can be enhanced by applying formal methods. In the past few years regulatory requirements for drug discovery systems have tightened. Due to this there is a need to introduce formal methods in the development lifecycle to prepare for future regulations.

When developing the control system Fillwell, we first depict the informal requirements with UML diagrams [BRJ99]. Since the initial specification should ensure safety and be proved to be consistent, the UML specification is translated into B Action Systems [BW98, WS98]. This translation can be done with the tool U2B [SB00]. The B Action Systems is a formalism for supporting the development of complex distributed systems. Since B Action Systems are Action Systems [BK83, BS96a] applied in the B Method [Abr96], we can benefit from the useful formalism for reasoning about distributed systems given by Action Systems and from the mechanised tool support of B [ClearSy03]. Using superposition refinement [BK83] we stepwise add more functionality to the specification and switch it into a more concrete and deterministic system. In each step, safety properties of the system are preserved. All the refinement steps are proved using the provers of Atelier B. We also give class and statechart diagrams for each development step. By first modelling the control system as a single entity, we can state safety properties of the entire system. At the end of the refinement process the system is then split into control system modules. That way we end up with the controlling software, *controller*, an environment that we want to control, *plant*, as well as their communication means, *actuators* and *sensors* [Sek98].

The traditional development at Wallac does not involve formal methods. However, since a few years the company uses UML [BRJ99] when designing systems. Hence, in order to gently introduce formal methods to Wallac we use a combination of UML and B. This combination is meant to facilitate the acceptance of the B method and formal methods in general and to give evidence for the benefits of using formal methods at Wallac, as well as to motivate the integration of formal methods into their regular development life cycle.

---

[1] EU-project MATISSE, IST-1999-11435, *http://www.matisse.qinetiq.com*

We first describe the healthcare case study in more detail in Section 2. In Section 3 we give an overview of the methodology used. We proceed with the description of the formal development in Section 4 and conclude in Section 5.

2. The healthcare case study

PerkinElmer Life Sciences (in this paper referred to as Wallac) designs, manufactures, develops, and markets analytical systems for use in drug discovery, mass population screening and other bioresearch and clinical diagnostics areas. In this case study the formal development is of a new Wallac's product – a workstation for preparing samples, Fillwell™ [PE01]. The workstation is shown in Figure 1. The system belongs to the class of products for drug discovery and bioresearch. The Fillwell microplate liquid handling workstation offers significantly advanced features in the line of the sample preparation systems. The Fillwell base unit consists of a dispense head dispensing liquid into microplates on a processing table. A gantry moves the dispense head with high precision and speed from one plate to another.

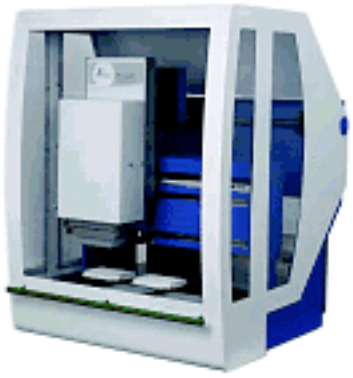


Figure 1: The Fillwell microplate liquid handling workstation.

The Fillwell workstation is the first liquid handling system specifically designed for high-density microplates. The system is modular and can be customised into a variety of configurations. The dispense head can have up to 384 tips attached via which the automated pipetting can be performed into plates with 96, 384 or 1536 wells. The head provides a precise dispensing with volumes from 0.5 to 300 μl. The processing table may contain up to 6 positions for plates, where three positions reside on an extension that is easily removable. In Figure 1 this processing table extension has been removed and there are only three plate positions. In order for the dispensing head to be able to reach all the positions on the processing table it is mounted on a gantry that can move horizontally and vertically. The precision of the gantry is very high with an accuracy of 100 μm. The main

application of the Fillwell workstation is drug discovery. Within this application area the system can be used for microplate replication, for dilution, transfer and addition of the liquid in the plates, for reformatting of plates with different densities (number of wells), as well as for rapid plate filling to homogenous and cell based systems. The system can function as a standalone workstation or be integrated into a robot.

The substances handled in the system can be extremely expensive (valued up to a billion EUR per kilogram). Moreover, the system may serve as part of an expensive production chain. The results of failures of such a system might lead to significant economical losses. Hence, the Fillwell system is both safety and money critical.

The setting of the case study. The healthcare case study was organised using an industry-as-laboratory approach. This means that the formal methods and B expertise is provided by the researchers at the academia and the domain knowledge is brought into the team by the experts at the R&D department of the industrial partner. Hence, while we in the team at Aabo Akademi performed the formal development of the workstation using UML and B, a team of software engineers at Wallac developed the workstation informally.

We met on a regular basis with the developers at Wallac during the development process. At these meetings the requirements of the Fillwell workstation were discussed to guide us in the formal development and the resulting documents of the formal development were analysed. The implementation of the informal design was performed componentwise and stepwise to learn about the behaviour of the instrument. This led to changes in the requirements as well as in the design. These changes were reflected in the formal development. However, the informal development was also affected by the formal development in the sense that topics discussed to solve the formal solution gave some new ideas for the informal development.

3. Overview of the methodology used

In the case study we use a methodology for the formal development that establishes an interface between a UML-based development process and safety analysis together with correctness proofs within B Action Systems [PTWBEJ01, PTW02].

3.1 Safety aspects of the development

The development process should ensure safety and reliability of the system under construction. The required dependability of the system can be achieved, only if safety and reliability attributes are considered from the early stages of the system development.

We conduct the software development hand-in-hand with the safety analysis. The safety analysis starts by identifying hazards [Sto96, Tro00] that are potentially dangerous in the

abstract specification and decide which methods are required to handle the hazards. While designing software for safety-critical systems, it is necessary to ensure that the suggested design does not introduce additional hazards. Moreover, we should ensure that the controlling software reacts promptly on hazardous situations by trying to force the system back to a safe state. The safety analysis proceeds by producing detailed descriptions of the hazards and finding the means to cope with them while stepwise refining the system.

3.2 UML-development incorporating safety aspects

UML (the Unified Modeling Language) is a graphical language for specifying, visualising, developing and documenting software-intensive systems [BRJ99]. Due to its scalability, UML is suitable for producing the initial specification of a control system [PS00]. The informal requirements of the system are depicted with UML diagrams. The functional requirements are captured together with their relationships in use case diagrams. Each use case expresses a service that the system will provide to a user. For example, the functional requirements of the Fillwell are to aspirate from and dispense liquid into plates, as well as to move the dispense head vertically and horizontally. 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 identified and grouped together into control system components in component diagrams. The component diagram is deduced from the use case diagram in such a way that each use case is mapped to a component service. For each component a class diagram is derived giving the attributes and methods of the component. The methods consist of the main functionality, i.e., the services, of the component as well as the abstract representation of errors and their possible fixes. The dynamic behaviour of the component is then specified with statechart diagrams. The informal specification of the system is given as a class diagram and a primitive statechart diagram as in Figures 2 and 3. We merely model state transitions and events causing these transitions at this level. We then gradually capture the details of the services in refined and more complex statechart diagrams.

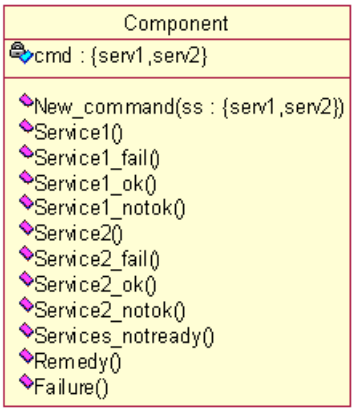


Figure 2: A class diagram of a component.

The transition *Service1* in the statechart diagram in Figure 3 models the execution of the command *serv1*, while *Service1_ok* refer to its successful result. Already in the initial specification we reserve a possibility for fault occurrence and system failure. The action *Service1_fail* models the failing to start the execution of the command *serv1*, while *Service1_notok* takes care of an unsuccesful execution of this command. There is also a possibility of spontaneous fault occurrence even when a service is not requested, as modelled by the action *Service_notready*. In all these failure transitions the system reacts on fault occurrence by entering state *Suspension*. From that state the system tries to execute a recovery procedure and continues functioning as specified by the transition *Remedy*. 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*. We model this with transition *Failure*.

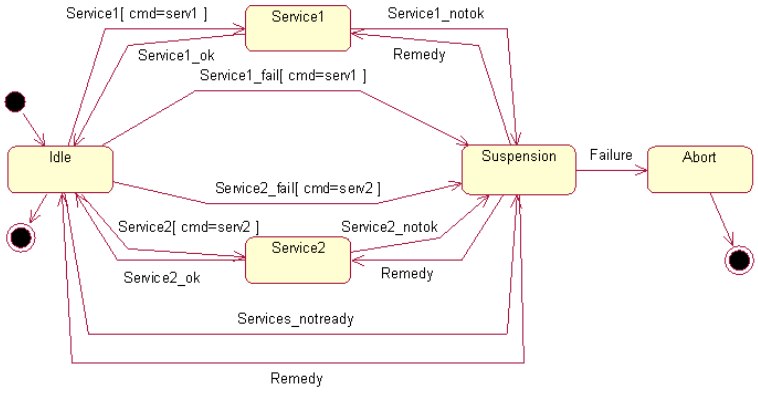


Figure 3: A primitive statechart diagram for a component.

3.3. B Action Systems in the development

In order to prove the consistency of the initial specification, we need a formal analysis tool. A formal method that comes with such tools is the B Method [Abr96]. We rely here on one of the tools supporting it, Atelier B [ClearSy03], during 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] which are based on the action systems formalism [BS96a] and related to Event based B [ClearSy03].

The abstract specification. The first task in our formal development is to create an abstract B Action System from the class diagram in Figure 2 and the statechart diagram in Figure 3. The tool U2B [SB00] supports this translation. The B Action System is identified by a unique name, *Component* below. The attributes/variables of the system are given in the **VARIABLES**-clause. In the basic statechart diagram the value of the attribute *cmd* corresponds to the services of the component, *Service1* or *Service2*. The attribute *state* models the state of the component. The types and the invariant properties of the local variables are given in the **INVARIANT**-clause and their initial values in the **INITIALISATION**-clause. The operations/services on the variables are given in the **OPERATIONS**-clause. Each transition in the statechart diagram corresponds to an operation in this clause.

```
MACHINE Component
VARIABLES
state, cmd
INVARIANT
state : {Idle,Service1,Service2,Suspension,Abort} ∧
cmd : {serv1,serv2}
INITIALISATION
state := Idle || cmd :: {serv1,serv2}
OPERATIONS
…
Service1 =          SELECT cmd = serv1 ∧ state = Idle THEN state := Service1 END;
Service1_fail =     SELECT cmd = serv1 ∧ state = Idle THEN state := Suspension END;
Service1_ok =       SELECT state = Service1 THEN state := Idle END;
Service1_notok =    SELECT state = Service1 THEN state := Suspension END;
…
Service_notready =  SELECT state = Idle THEN state := Suspension END;
Remedy =            SELECT state = Suspension THEN state :: {Idle,Service1,Service2} END;
Failure = SELECT state = Suspension THEN state := Abort END
END
```

With B Action Systems 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.

We can also declare local and global *procedures* in B Action Systems. The procedures are discussed in more detail elsewhere [SW00, Wal98].

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 the *superposition refinement* method [BK83, BS96b], where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behavior. This stepwise introduction of implementation details is especially convenient when dealing with complex control systems. 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. For each refinement step we create class diagrams and statechart diagrams as well as B refinement machines. The tool U2B assists in the process of writing refinements in UML and translate them to B code [SW02, STW03].

As the system development proceeds we obtain more elaborated information about faults and conditions of failure occurrence. The refinement step introduces a distinction between faults. The operation *Service1_fail* models fault resulting from an attempt to provide a service from an incorrect initial state. This situation might be caused by faults occurred previously or by a logical error in the calling command. For example, in the Fillwell the dispense head might be too high up to dispense liquid in a safe manner. The operation *Service1_notok* models fault occurrence during the execution of the action. These kind of faults are caused by the physical failures of the system components involved in the execution, e.g., the Fillwell dispense head does not reach its destination. We also introduce a distinction between different repair procedures by adjusting the *Remedy* operation for each fault accordingly.

Using Atelier B we can formally prove that the refinement is sound. A number of proof obligations [BW98, WS98] can be generated automatically by Atelier B with the help of the translator Evt2b [ClearSy03]. These proof obligations can be discharged using the autoprover and the interprover in Atelier B. With the proof obligations that are generated for a refinement it is checked 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 translator Evt2b proof

obligations on the non-divergence of the auxiliary operations can be generated. With the help of the translator Evt2b we can also check 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 the actions should enable it. The error detection in the system should find at least the same erroneous situations in the refinement as in the specification [Tro00].

3.4 Control systems development

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* [Sek98, PRTWJ01]. The plant 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. To obtain such a decomposition in B Action Systems, the operations in the machines to be decomposed have to be split, and the variables are partitioned among the plant and the controller machines. As for the splitting of operations, each operation of the form

Oper = **SELECT** state=act1 ∧ A **THEN** B; state :=act2 **END**

is replaced with the following operation in the plant

Oper' = **SELECT** state=act1 ∧ A **THEN** Act2 **END**,

where the procedure *Act2* of the controller is

Act2 = **PRE** state=act1 ∧ A **THEN** B; state:=act2 **END**.

Obviously, the effect of the new operation *Oper'* is the same as the effect of the old operation *Oper*. Adding procedures and keeping the old functionality agrees with the superposition refinement step.

In the last development step, we determine the sensors and the actuators for the components from the controller and plant specifications. 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. In B Action Systems, the actuators and sensors are global variables of the plant and the controller, and these variables are put into separate machines. The sensor variables are set by the plant and read by the controller, while the actuator variables are set by the controller and read by the plant. Due to this, the plant **INCLUDES** the sensor variables and **SEES** the actuator variables. Dually, the controller **INCLUDES** the actuator variables and **SEES** the sensor variables. These refinement steps merely involve rewriting and restructuring. Hence, the final control system is the result of a correct formal development process.

4. The formal Fillwell development

In this section we show the development of the Fillwell, the microplate liquid handling workstation, using the methodology above. We first give the requirements of the system and then proceed with the actual development. The complete development can be found on the homepage of the formal Fillwell [FormFill03].

4.1 The requirements

The Fillwell workstation consists of three parts, a dispense head, a gantry and an operating table. The dispense head is dispensing liquid into and aspirating liquid from microplates on the operating table with high precision. The head is attached to a gantry which moves it horizontally and vertically over the operating table. The precision of the gantry should also be very high. The user performs experiments with the Fillwell by loading a protocol to the system. The protocol is then interpreted and executed by the Fillwell system. The protocol contains commands for the gantry to move the dispense head vertically and horizontally, and for the dispense head to aspirate and dispense a certain amount of liquid or air, as well as to wash the tips.

The operating table has three positions for plates. An additional part with three plate positions can be added to the table. Each plate position may contain a plateholder, a tipwasher unit or a constant level reservoir. A plateholder in turn may contain plates with 96, 384 or 1536 wells. The dispense head can have up to 384 tips attached. When the head has 96 tips attached, it can operate on plates with 96 or 384 wells and with 384 tips the head can operate on plates with 384 or 1536 wells. The tips are arranged in rows and columns, 8 x 12 (96) or 16 x 24 (384). The dispense head may have only a row of tips attached, i.e., 12 of 96 or 24 of 384, or a column of tips, i.e., 8 of 96 or 16 of 384. The amount of liquid in the tips and in each well should be registered. If a tip contains both air and liquid the air is always above the liquid. Hence, air cannot be aspirated into a tip, if there is liquid in the tip. When dispensing air the head should preferably not be beneath the liquid level.

In order for the dispensing head to be able to reach all the plate positions on the operating table it is mounted on a gantry that can move it vertically and horizontally. The vertical movement (movement along the z-axis) has three reference points: the height of the plate, the liquid level and the bottom level of the plate. An offset value should state exactly where the lower ends of the tips are positioned in relation to the given reference point. The gantry moves the head horizontally over the table in x- and y-directions to a certain plate. When the head is moving over the table it should be positioned high enough for the tips not to touch the plates. The head can be placed above each of the plates on the operating table. Since the head may have only a row or a column of tips attached, it is not always placed above a plate so that the tip in the front left corner of the dispense head is right above the well in the front left corner of the plate. Internal positions give the exact

location of the dispense head above a plate. Initially, the dispense head is placed in a home position above the table in its back left corner.

The Fillwell workstation has three lamps (green, yellow and red) indicating movement and error situations. The green lamp indicates that power is on in the Fillwell workstation. The yellow lamp indicates that the dispense head is allowed to move. When an error has occurred the red lamp is switched on. The Fillwell also has a pause button and an emergency button to enable the user to stop the system at any time during an execution. When the pause button is pressed once, the execution of the system should stop. When the button is pressed once again, the system should continue the execution from the state it was stopped. When the emergency button is pressed, the execution should terminate and the red lamp should be switched on.

The following *safety requirements* should hold for the Fillwell:

- When the dispense head is moving horizontally over the table it should be above all accessories (plates, reservoirs, etc.) placed on the operating table.
- The yellow lamp must be switched on when the dispense head is moving.
- The dispense head is not allowed to dispense or aspirate when moving.
- The head must not move beyond its upper and lower end position.

4.2 The structure of the Fillwell development

The formal Fillwell specification is partitioned into three components: Dispenser, XYZ-Driver and Protocol runner. The component Dispenser models the dispense head and XYZ-Driver models the gantry moving the head. Protocol runner reads a protocol given by the user and performs the commands defined in it. The components XYZ-Driver and Dispenser share data about accessories. The shared data constitute a separate component – Operating table. The overall structure of the formal Fillwell specification is given in Figure 4. The same component partitioning has been used also in the informal development.

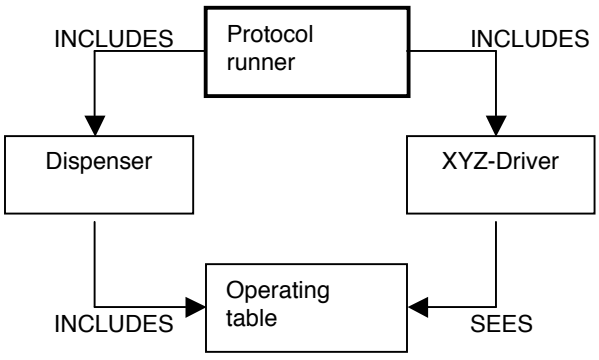


Figure 4: The structure of the formal Fillwell development.

The component Dispenser is described in Section 4.3. It changes the amount of liquid in the plates and, hence, the data of Operating table. The component XYZ-Driver, described in Section 4.4, reads the heights of all the accessories from the component Operating table in order to be able to move across the table without damaging the tips attached to the dispensing head. The component Protocol runner, described in Section 4.5, co-ordinates the execution of Dispenser and XYZ-Driver.

During the refinement of the system concrete information about the accessories is added in a stepwise manner to Operating table. Even though Dispenser and XYZ-Driver are developed independently of each other, their refinement steps are co-ordinated via the refinements of the component Operating table.

4.3 The component Dispenser

We first describe the development of the comoponent Dispenser that aspirate and dispense air and liquid. Dispenser changes the amount of liquid in the plates on the operating table. Hence, the component Operating table is developed hand-in-hand with the component Dispenser. When we refine Dispenser by stepwise adding new features to it, we add the corresponding features to the component Operating table.

4.3.1 The specification

We start the development by defining the services of Dispenser. According to the requirements the services are:

- to aspirate liquid from an accessory,
- to dispense liquid into an accessory,
- to aspirate air, and
- to blow (or dispense) air.

Each service is initiated by a call from Protocol runner. The operations of Dispenser are then executed until the service is completed or an error has occurred. For each service we take into consideration the possible errors and the remedies that could fix the erroneous situation.

In this report we describe the service aspirate liquid in more detail. The three other services; aspirate air, dispense liquid and dispense air, can be specified in a similar way. First we give the typical course of event for Dispenser *aspirating liquid* from an accessory:

1. The protocol contains a command to aspirate *amt* units of liquid.
2. System checks that the accessory is either a *plate* or a *tip washer*; if wrong accessory then AF1.

3. System checks the position of the operating head; if not within the accessory then AF2.
4. System checks that adding *amt* units of liquid to the tips does not exceed the tip capacity, *tip_capacity*; if it does then AF3.
5. System checks amount of liquid in the accessory; if there is not enough liquid to aspirate *amt* units of liquid then AF4.
6. Head aspirates *amt* units of liquid.
7. System checks for error message from dispensing head module; if head module failed then AF5.
8. System reports success of execution of the aspiration to the calling protocol.

Error reports:

AF1. Wrong type of accessory under the dispense head. The accessory is not a *plate* or a *tip washer*.
Remedy: User finds out the reason for the failure and resumes or aborts the calling protocol. In case of resuming user manually changes the accessory under the dispense head after moving up the head.

AF2. The dispense head, i.e., the head position sensor reading, is not within the accessory.
Remedy: User finds out the reason for the failure situation and resumes or aborts the calling protocol. In case of resuming user manually invokes the procedure move to a position *zpos* that is within the accessory, and resumes protocol execution.

AF3. Adding *amt* units of liquid to the tips exceeds the tip capacity.
Remedy: User finds out the reason for the failure situation and resumes or aborts the calling protocol. In case of resuming user manually changes parameter *amt* and resumes calling protocol execution.

AF4. Accessory contains less liquid than needed to aspirate, *amt* units.
Remedy: User finds out the reason for the situation and resumes or aborts the calling protocol. In case of resuming user manually gives command to add liquid and resumes protocol.

AF5. Error message from dispense head module detected.
Remedy: User initiates a maintenance procedure to fix the dispense head error. After maintenance user resumes or aborts protocol.

The statechart diagram of the abstract specification of the service aspirate liquid in Dispenser is shown in Figure 5. Each transition in Figure 5 corrsponds to an operation in B. When the state of Dispenser is idle, *didle*, and the command from Protocol runner is aspirate liquid (*NewAspirateLiqCommand*) the state is changed to *dprep*, to mark that Dispenser is preparing to execute the command. The command of Dispenser is assigned *aspl*, ***asp***irate ***l***iquid. The parameter of the command, *p_amt*, states how much liquid is to be aspirated. In case there is something wrong with the operating table, the tips or the operating head, *ServiceNotReady* changes the state of Dispenser to suspended, *didle_susp*. When Dispenser is in state *dprep* and the command is *aspl* to aspirate liquid, it proceeds with the operation *AspirateLiq* that changes the state to *daspl* to indicate that

Dispenser has aspirated liquid. If something is wrong, e.g., the setting of the instrument is not correct, the operation *AspirateLiqFail* changes the state to suspended, *dsusp*. When Dispenser is in state *daspl*, the command is *aspl* and the aspiration has been performed correctly, operation *AspirateLiqOk* is enabled and completes the aspirate liquid service changing the state back to *didle*. If something went wrong when aspirating liquid, operation *AspirateLiqNotOk* suspends the service, i.e. changes the state to *dsusp*. A possible recovery from the suspended states is via the operations *Remedy*, *AspirateLiqPrepRemedy* and *AspirateLiqRemedy*, respectively. If there is no operation that can fix the erroneous situation, the operation *Failure* changes the state to *dabort* and aborts the service.

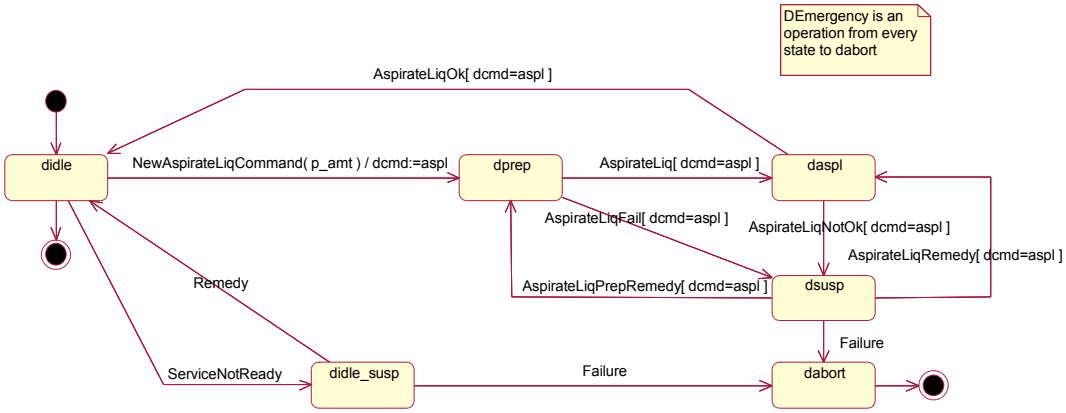


Figure 5: Abstract statechart diagram of the operation *aspirate liquid*.

4.3.2 Feature 1: General liquid and accessory information

As a first feature we add information about the accessories on the operating table and the liquid to be analysed. First we only state whether there is an accessory or not at a specific position on the operating table with the variable *AccessoryPresent*. We model the lowest possible height at which the dispense head may move over the table with the variable *zmid*. Furthermore, we are interested in how deep the dispense head may go into an accessory. We state this with the variable *zminAllowed*. The position of the dispense head is modeled with the variable *ZCoord*. The presence of liquid in the accessories is modeled with a Boolean variable for each accessory position, *Accessory_LiquidPresent*. The tips attached to the dispense head may also contain liquid and/or air. This is modeled using the two Boolean variables *Tips_LiquidPresent* and *Tips_AirPresent*.

In this step the state *dsusp* is partitioned into a number of new states, one for each service, in order to be able to create more accurate remedy operations. Due to the new suspension states we rename the state variable *dstate1* and give its relation to *dstate* in the refinement invariant.

In the invariant we state the relationship between the new variables.

Accessory_LiquidPresent(extpos) = TRUE ⇒ AccessoryPresent(extpos) = TRUE
∧ dstate1=daspl1 ⇒ Tips_Liquid_Present = TRUE
∧ dstate1=daspa1 ⇒ Tips_Air_Present = TRUE
∧ dstate1=ddispl1 ⇒ Accessory_LiquidPresent(extpos) = TRUE

Intuitively, this invariant states that if there is liquid present in an accessory at a position *extpos*, then there has to be an accessory at that position. Additionally, if Dispenser has aspirated liquid (*dstate1 = daspl1)*, then there has to be liquid in the tips. Correspondingly, if Dispenser has aspirated air *(dstate1 = daspa1)*, then there has to be air in the tips. If on the other hand Dispenser has dispensed liquid (*dstate1 = ddispl1)*, then there has to be liquid in the accessory. The complete invariant can be found in the B machine of the first refinement of Dispenser on the homepage of the formal Fillwell [FormFill03].

Let us now take a closer look at the refined service *aspirate liquid*. Liquid can be aspirated when there is an accessory containing liquid under the dispense head, i.e., at the position *extpos*, and the head is within the allowed range, which is stated by the guard:

AccessoryPresent(extpos) = TRUE ∧ Accessory_LiquidPresent(extpos) = TRUE
∧ ZCoord ∈ zminAllowed(extpos)..zmid.

In Appendix A.1 the statechart diagram of the refined service *AspirateLiq* is given. The decision symbols in the diagram are used to split up the guards of the operations. Hence, a guard of an operation is the conjunction of all the guards on the transitions from one state to another via the decision symbols. In this way the failure operations can be conveniently modeled as exceptions to the service.

For the other services, *aspirate air*, *dispense liquid* and *dispense air*, the guards are similar to the guard of the operation aspirate liquid. We can *aspirate air* if there is no liquid in the tips and the operating head is high enough above the accessory

Tips_LiquidPresent = FALSE ∧ ZCoord > zminAllowed(extpos).

We can *dispense liquid*, if there is an accessory under the dispense head, the tips contain liquid and the head is inside the accessory

AccessoryPresent(extpos) = TRUE ∧ Tips_LiquidPresent = TRUE
∧ ZCoord ∈ zminAllowed(extpos)..zmid.

We can *dispense air* if there is air, and only air in the tips, and the head is high enough above the accessory

Tips_LiquidPresent = FALSE ∧ Tips_AirPresent = TRUE ∧ ZCoord > zminAllowed(extpos).

The statechart diagrams of the services *aspirate air*, *dispense liquid* and *dispense air* are given on the homepage of the formal Fillwell [FormFill03].

4.3.3 Feature 2: Detailed liquid information

In the second refinement step of Dispenser we introduce the types of the accessories on the operating table. An accessory may be of different types, e.g., a plate holder, a plate, a tip washer or a constant level reservoir (represented as a plate with one well), which is represented by the variable *AccessoryType*. The relationship between *AccessoryType* and *AccessoryPresent* is stated in the invariant as:

AccessoryType(extpos) ≠ empty ⇒ AccessoryPresent(extpos) = TRUE
∧ AccessoryType(extpos) = empty ⇒ AccessoryPresent(extpos) = FALSE

Intuitively, this means that if the type of the accessory at position *extpos* is a plate, a tip washer or a plate holder, i.e. it is not empty, then there is an accessory at the position *extpos*. If the type of accessory at position extpos is empty, then there is no accessory present at the position *extpos*.

When we introduce more detailed information about the liquid in the tips of Dispenser, we have to introduce corresponding features to the component Operating table. A plate should always be placed on a plate holder on the operating table. Hence, we introduce the variable *Plateholder* as a function from the position on the operating table to a record of two fields stating whether a plate is present on the plate holder, *plate_present*, and giving the height of the plate holder, *height.*

Plateholder ∈ 1..max_extpos +-> struct(plate_present ∈ BOOL, height ∈ NAT)

Correspondingly, the plate is represented as a function from the plate positions to a record of five fields; plate id, height, liquid amount, minimum height and maximum volume.

Plate_1 ∈ 1..max_extpos +->
struct(pid ∈ PID, height ∈ NAT, liquid_amt ∈ NAT, minheight ∈ NAT, maxVol ∈ NAT)

Every plate has a unique identifier, *pid*. The variable *height* gives the height of the plate and the minimum height, *minheight*, is the distance from the bottom of the well of the plate to the surface of the operating table. The variable *liquid_amt* represents the amount of liquid on the plate and the maximum volume, *maxVol*, represents the maximum amount of liquid the plate can hold. The invariant of the plate machine states that the liquid amount always should be less or equal to the maximum volume.

∀xx.(xx ∈ 1..max_extpos ∧ xx ∈ dom(Plate_1) ⇒ (Plate_1(xx))'liquid_amt ≤ (Plate_1(xx))'maxVol)

If there is a plate present on the plate holder at any position of the operating table the accessory type of that position has to be a plate.

∀xx.(xx ∈1..max_extpos ∧ (Plateholder(xx))'plate_present = TRUE ∧ xx ∈ dom(Plateholder)
⇒ xx ∈ dom(Plate_1) ∧ AccessoryType(xx) = plate)

Also the tipwasher is represented as a function from the positions on the operating table to a record of five fields; plate id, height, minimum height, liquid amount and maximum volume.

Tipwasher ∈ 1..max_extpos +->
struct(wid ∈ WID, height ∈ NAT, minheight ∈ NAT, liquid_amt ∈ NAT, maxVol ∈ NAT)

Every tip washer has a unique id, *wid*. The variable *height* gives the height of the tip washer and the minimum height, *minheight*, is the distance from the bottom of the tip washer to the surface of the operating table. The liquid amount, *liquid_amt*, represents the amount of liquid in the tip washer and the maximum volume, *maxVol*, represents the maximum amount of liquid the tip washer can hold. The invariant states that the liquid amount should always be less or equal to the maximum volume.

∀xx.(xx ∈1..max_extpos ∧ xx ∈ dom(Tipwasher)
⇒ (Tipwasher(xx))'liquid_amt ≤ (Tipwasher(xx))'maxVol)

In this refinement step the amount of liquid in an accessory (plate or tip washer) is stated with the variable *AccessoryLiquidAmt*. In the invariant we state that if the accessory at position *extpos* is a plate then *AccessoryLiquidAmt* is equal to the amount of liquid in the plate at that position. On the other hand, if the accessory is a tip washer then *AccessoryLiquidAmt* is equal to the amount of liquid in the tip washer.

AccessoryType(extpos)=plate ⇒ AccessoryLiquidAmt(extpos) = (Plate_1(extpos))'liquid_amt
∧ AccessoryType(extpos)=tipwasher⇒ AccessoryLiquidAmt(extpos)=(Tipwasher(extpos))'liquid_amt

The invariant also states that if the amount of liquid in the accessory at position extpos is 0, then there is no liquid in that accessory. If the amount is greater than 0, then there is liquid in the accessory.

AccessoryLiquidAmt(extpos) = 0 ⇔ Accessory_LiquidPresent(extpos) = FALSE
∧ AccessoryLiquidAmt(extpos)> 0 ⇔ Accessory_LiquidPresent(extpos) = TRUE

Moreover, the exact amount of liquid and air in the tips is of interest at this step. This is represented with the variables *Tips_Liquid_Amt* and *Tips_Air_Amt*. The capacity of the tips is represented by a variable *tip_cap*. The total amount of liquid and air in the tips may not exceed the capacity of the tips.

(Tips_Liquid_Amt + Tips_Air_Amt) ∈ 0..tip_cap

The height of each accessory, *AccessoryHeight*, and how deep the dispense head may go into the accessory, *AccessoryMinheight*, are also considered in this step. The minimum accessory height of a position is always less or equal to the height of the accessory at that position.

∀xx.(xx ∈ 1..max_extpos ⇒ AccessoryMinheight(xx) ≤ AccessoryHeight(xx))

The lowest height at which the dispense head is allowed to move across the operating table is indicated in the system using a variable *xymoveOk*. This variable is equal to the maximum height of all the accessories on the operating table.

max(ran(AccessoryHeight)) = xymoveOk

The operations are refined taking into account these new variables. The guard of the operation to aspirate liquid is shown below:

AccessoryType(extpos) ∈ {plate, tipwasher}
∧ ZCoord ∈ AccessoryMinheight(extpos)..AccessoryHeight(extpos)
∧ Tips_Liquid_Amt+Tips_Air_Amt+amt1 ≤ tip_cap
∧ AccessoryLiquidAmt(extpos) ≥ amt1

There has to be a plate or a tip washer under the dispense head in order to aspirate liquid. The head also has to be within the accessory. Furthermore, the tip capacity is not allowed to be exceeded with the amount of air and liquid in the tips and the amount, *amt1*, of liquid to be aspirated. Finally, there has to be enough liquid in the accessory to aspirate *amt1* units of liquid. When the operation to aspirate liquid is enabled, the liquid amount in the accessory is decreased by *amt1* units of liquid and the liquid amount in the tips is increased correspondingly. The refined statechart diagram of the service can be found in Appendix A.2.

We can *aspirate air* if there is no liquid in the tips, the operating head is high enough above the accessory and the tip capacity is not exceeded.

Tips_Liquid_Amt = 0 ∧ ZCoord > AccessoryMinheight(extpos) ∧ Tips_Air_Amt+amt1 ≤ tip_cap

When the operation is enabled, *amt1* units of air are added to the amount of air in the tips.

We can *dispense liquid*, if there is a plate or a tip washer under the dispense head, the tips contain enough liquid to dispense *amt1* units, the head is within the accessory and the accessory capacity is not exceeded.

AccessoryType(extpos) ∈ {plate, tipwasher} ∧ Tips_Liquid_Amt ≥ amt1
∧ ZCoord ∈ AccessoryMinheight(extpos)..AccessoryHeight(extpos)
∧ ((AccessoryType(extpos)=plate ∧ (Plate_1(extpos))'liquid_amt+amt1 ≤ (Plate_1(extpos))'maxVol)
  ∨ (AccessoryType(extpos)=tipwasher
                ∧ (Tipwasher(extpos))'liquid_amt+amt1 ≤ (Tipwasher(extpos))'maxVol)).

When the operation is enabled, the liquid amount in the accessory is increased by *amt1* units of liquid and the amount of liquid in the tips are decreased correspondingly.

Finally, we can *dispense air*, if there is enough air in the tips in order to dispense *amt1* units, there is no liquid in the tips and the dispense head is above the bottom of the accessory.

Tips_Air_Amt ≥ amt1 ∧ Tips_Liquid_Amt =0 ∧ ZCoord > AccesoryMinheight(extpos)

When the operation is enabled *amt1* units of air is blown out from the tips. The statechart diagrams for the services *aspirate air*, *dispense liquid* and *dispense air* are similar to the one for service *aspirate liquid* [FormFill03].

4.3.4 Feature 3: Detailed plate and tips information

In the third refinement step we add features concerning the types of plates on the operating table. The number of wells the plate contains gives the type of a plate. A plate may contain 1, 96 (12 columns x 8 rows), 384 (24 columns x 16 rows) or 1536 (48 columns x 32 rows) wells. The wells on a plate are numbered from 1 to the number of wells on the plate as in Figure 6. The first row of a 96-well plate has the wells numbered 1 to 12, the second row is numbered 13 to 24 and so on.

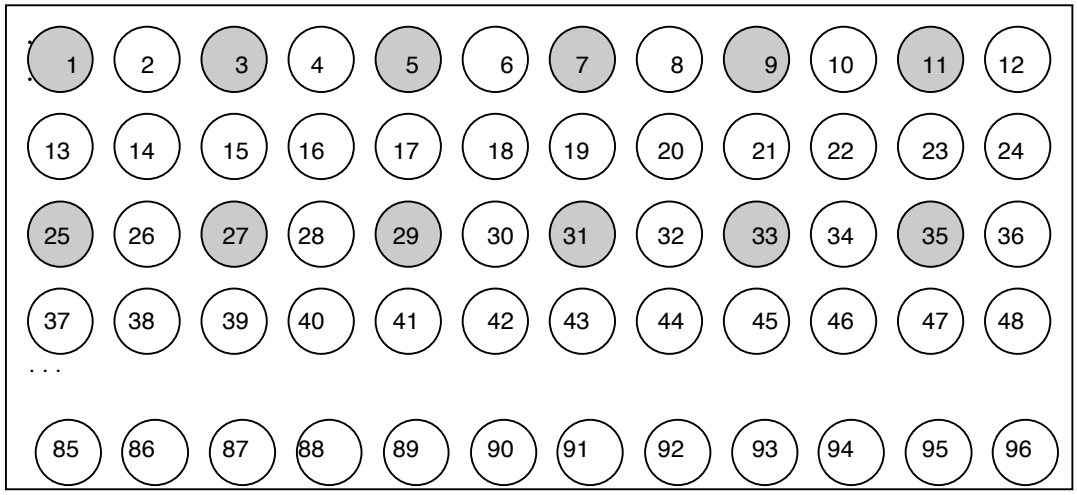


Figure 6: A dispense head with 24 tips in internal position 1 over a 96-well plate.

Since we consider the number of wells on the plate in this refinement step, the number of tips attached to the dispense head also becomes essential. The dispense head can have a block of 96 or 384 tips attached. However, all the tips of the block do not have to be attached. There could be only a single tip or alternatively a column of tips (8/96, 16/384) or a row of tips (12/96, 24/384). If we have a block of 96 tips over a plate with 384 wells the tips go into every second well of every second row of the plate. The same holds for a block of 384 tips over a plate with 1536 wells. Since the number of tips is not always the same as the number of wells, we need to register which of the wells in a plate that will be affected from dispensing and aspirating. The variable *wellset* keeps track of this.

If the number of wells differs from the number of tips, the dispense head can be positioned in a number of internal positions, *intpos*, of the plate. For example, if we have 8 tips and a 96-well plate, then the tips can be positioned in 12 different internal positions. The possible values of the internal position *intpos* are given in Table 1 below.

\ noOfTips noOfWells\	1	8	12	16	24	96	384
1	1	1	1	1	1	1	1
96	1..96	1..12	1..8	-	-	1	-
384	1..384	-	-	1..24	1..16	1..4	1
1536	-	-	-	-	-	-	1..4

Table 1: The internal positions for all possible plates.

The variable *wellset* can easiest be illustrated with Figure 6. In order to easier be able to show the set of wells graphically we have chosen the fictive combination of 96 wells and 24 tips with 4 internal positions. Note that 24 tips above 96 wells is not possible in Fillwell. For internal position 1 the tips are above wells 1, 3, 5, 7, 9, 11, 25, 27, 29, … , 83. For position 2 the tips are above wells 2, 4, 6, 8, 10, 12, 26, 28, 30, … , 84 and so on. The wellset for internal position 1 are the dark coloured wells in Figure 6. These are the wells affected by an aspirate or a dispense command.

The amount of liquid per well becomes of interest in this refinement step. This is stated with the variable *liquid_amt2*. The relationship between the old variable *liquid_amt* and the new *liquid_amt2* states that the sum of the amount of liquid in all the wells of a plate (*Plate_2*) is the same as the liquid amount on the plate (*Plate_1*).

(Plate_1(extpos))'liquid_amt =
 (yy).(yy 1..(Plate_2(extpos))'noOfWells | ((Plate_2(extpos))'liquid_amt2)(yy))

Since we here consider the amount of liquid per well, we also have to introduce the maximum volume of each well, instead of the maximum volume of the whole plate.

(Plate_1(extpos))'maxVol = (Plate_2(extpos))'noOfWells * (Plate_2(extpos))'maxVol.

Correspondingly we consider the liquid amount in each tip, *Tips_Liquid_Amt2*. Even if not all the wells need to contain the same amount of liquid, the amount of air and liquid in the tips is the same for all the tips. The sum of the amount of liquid in each tip, *Tips_Liquid_Amt2*, is the same as the amount of liquid in all the tips, *Tips_Liquid_Amt*.

Tips_Liquid_Amt = Tips_Liquid_Amt2 * tips1'numberOfTips

The amount of air in the tips is represented in a similar way. Due to this fact, the capacity of the tips is redefined to correspond to the capacity of a single tip and represented by a variable *tip_capacity*. The capacity of all the tips together is the same as the value of the old tip capacity *tip_cap*.

tips1'tip_capacity * tips1'numberOfTips = tip_cap

The new features are reflected in the operations. The operation *AspirateLiq* of the service aspirate liquid is enabled when the accessory is a plate or a tipwasher that contains

enough liquid for each tip to aspirate *amt2* units of liquid. The tips of the dispense head also have to be within the well and the capacity of the tips should not be exceeded.

AccessoryType(extpos) ∈ {plate,tipwasher}
∧ ((AccessoryType(extpos) = plate
∧ ∀yy.(yy ∈ (wellset(extpos))(intpos) ⇒ ((Plate_2(extpos))'liquid_amt2)(yy) ≥ amt2))
∨ (AccessoryType(extpos)=tipwasher
∧ (Tipwasher(extpos))'liquid_amt ≥ amt2 * tips1'numberOfTips))
∧ ZCoord ∈ AccessoryMinheight(extpos)..AccessoryHeight(extpos)
∧ Tips_Liquid_Amt2 + Tips_Air_Amt2+amt2 ≤ tips1'tip_capacity

When the operation is enabled the amount of liquid is decreased with *amt2* units from the wells beneath the tips and the amount of liquid in the tips is increased correspondingly. The refined statechart diagram of the service can be found in Appendix A.3.

For the operation *aspirate air* to be enabled, the amount of liquid in every tip has to be 0 and the capacity cannot be exceeded. The tips should either be inside or above the wells.

Tips_Liquid_Amt2 = 0 ∧ Tips_Liquid_Amt2 + Tips_Air_Amt2 + amt2 ≤ tips1'tip_capacity
∧ ZCoord > AccessoryMinheight(extpos)

The operation *dispense liquid* is enabled when each well in the plate or the tipwasher has enough capacity to receive the dispensed liquid. We also have to make sure that the head with the tips is within the wells or the tipwasher.

AccessoryType(extpos) ∈ {plate, tipwasher}
∧ ((AccessoryType(extpos) = plate
∧ ∀yy.(yy ∈ (wellset(extpos))(intpos) ⇒
((Plate_2(extpos))'liquid_amt2)(yy) + amt2 ≤ (Plate_2(extpos))'maxVol))
∨ (AccessoryType(extpos) = tipwasher
∧ (Tipwasher(extpos))'liquid_amt + tips1'numberOfTips * amt2 ≤ (Tipwasher(extpos))'maxVol))
∧ ZCoord ∈ AccessoryMinheight(extpos)..AccessoryHeight(extpos)

When the operation is enabled the amount of liquid in the tips is decreased with *amt2* units and the liquid amount of the wells under the tips is increased correspondingly.

The operation *dispense air* is enabled when there is enough air in the tips, but no liquid. The tips of the dispense head could be in the wells or above them.

ZCoord > AccessoryMinheight(extpos) ∧ Tips_Liquid_Amt2 = 0 ∧ Tips_Air_Amt2 ≥ amt2

When the operation is enabled, *amt2* units of air is blown out from the tips.

4.3.5 Control system development

The component Dispenser is rewritten as a plant and a controller. All the variables of Dispenser and Operating table are considered to be actuators of Dispenser. This step is straight forward and therefore not shown here. The B machines of the plant and controller for Dispenser can be found on the Formal Fillwell homepage [FormFill03].

4.3.6 Proving correctness

The consistency of the specification as well as the correctness of each refinement step is proved with the tool Atelier B. The proof obligations generated for each step are of the form described in subsection 2.4.3. In Table 2 we give the number of obvious and generated proof obligations at each refinement step, as well as the number of B machines and the number of lines of code for each step. The B machines for the component Operating table are included in Table 2.

Dispenser	Machines	Lines	Obvious p.o.	Generated p.o	Autoproved	Percent
spec	**2**	**430**	**262**	**202**	**202**	**100 %**
ref. step 1	**7**	**940**	**659**	**1 600**	**1 599**	**100 %**
ref. step 2	**11**	**1 610**	**3 523**	**812**	**742**	**91 %**
ref. step 3	**11**	**2 150**	**4401**	**255**	**212**	**91 %**

Table 2: Quantitative information on the development of Dispenser.

We can note that refinement step 2 introduce the most new B machines and new lines of code. The obvious proof obligations are generated by the tool and are discharged immediately. The user needs to prove the generated proof obligations with the proof tool. Preferably as many of the generated proof obligations as possible should be discharged automatically. For the first refinement step we have a good result of 100%. The second and third refinement steps introduce many quantified expressions in the invariant, which in turn leads to proof obligations that are difficult for the tool to discharge automatically. These proof obligations have, however, been discharged interactively.

4.4 The component XYZ-Driver

Let us now consider the development of the component XYZ-Driver. It moves the dispense head vertically and horizontally over the operating table. In order to be able to perform the moving, XYZ-Driver needs to be able to read the variables of the component Operating table.

4.4.1 The specification

The component XYZ-Driver provides two services; to move the dispense head horizontally (in x- and y-directions) and vertically (in z-direction) over the operating table. The component XYZ-Driver performs the moving with the help of three components X-Driver, Y-Driver and Z-Driver. X-, Y- and Z-Driver each handles the low-level aspects of the moving in the direction indicated by the name of the component. The XYZ-Driver co-ordinates these drivers in such a way that Z-Driver must be idle when X-

and Y-Driver are active and vice versa. An overview of the XYZ-Driver component is given in Figure 7.

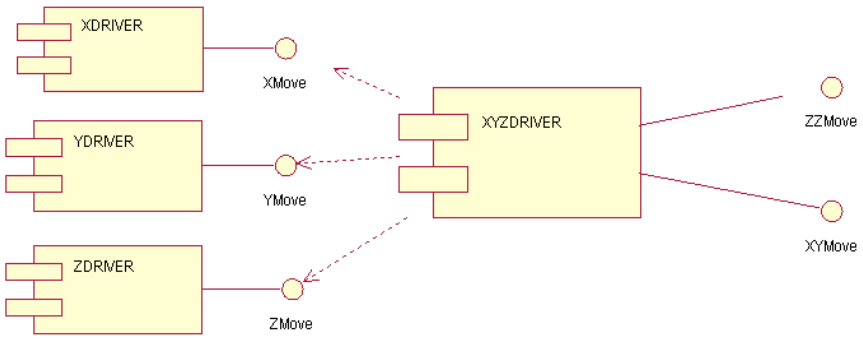


Figure 7: Component diagram for XYZ-Driver

We first study the X-Driver specification. The possible failure situations are taken into consideration, as well as the remedies that could fix the errors. The typical course of events for moving in the x-direction is given below. Moving in y- and z-directions are dealt with in a similar manner.

1. The operation *XMove* of X-Driver is called by XYZ-Driver.
2. System reads the input parameter *pos*, the desired position of the dispense head after the move, and *xspeed,* the speed to move with.
3. System checks that the yellow lamp is switched on; if it is not, then XF1.
4. The component X-Driver moves to x-position *pos* with speed *xspeed*.
5. System checks if the current position is *pos*; if not, then XF2.

Failure reports:

XF1. Yellow lamp is not switched on.
Remedy: User repairs the lamp and resumes or aborts the calling protocol.

XF2. X-Driver has not reached position *pos*.
Remedy: User calibrates the position sensors and resumes or aborts the calling protocol execution.

The service for moving in the x-direction is given as an abstract statechart diagram in Figure 8. The moving is modeled with the operation *XMove* that changes the state of the X-Driver from *idle* to *xmove*. If the move was successful, the operation *XMoveOk* is enabled and takes X-Driver back to state *idle*. On the other hand if the move was not successful, the X-Driver enters the state *xsuspended.* A remedy operation is enabled, if it is possible to recover from the failure. It will change the state of X-Driver back to the

state it was in before the suspension. If there are no possible remedies, X-Driver will abort execution with the operation *XFailure*. The statechart diagrams for the components Y- and Z-Driver are similar and are not shown here. They can be found together with the B-specification of the component on the Formal Fillwell homepage [FormFill03].

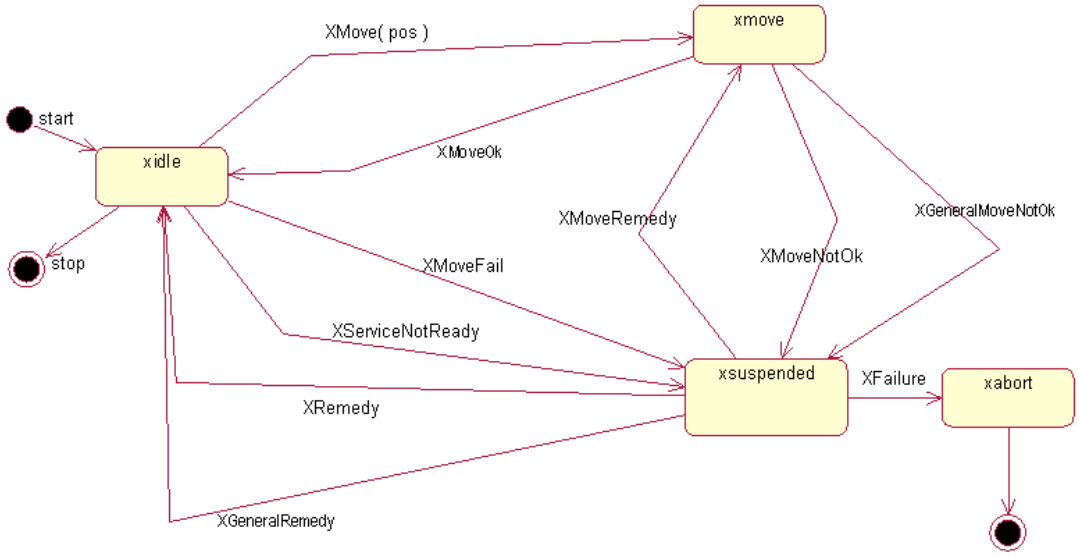


Figure 8: Abstract statechart diagram of X-Driver.

Let us now study the component XYZ-Driver that coordinates the moving performed by X-, Y- and Z-Driver. As for the component Dispenser the services of XYZ-Driver are initiated by a command from Protocol runner. When the XYZ-Driver receives the command to move horizontally, it first moves the dispense head vertically to a height at which it can freely move over the table. The yellow lamp should be switched on when the dispense head is moving. The typical course of events description for the horizontal moving, *xymove*, is as follows.

1. The component XYZ-Driver receives command *xymove* from Protocol runner to move with a certain speed to position (*extpos, intpos*), the accessory position on the operating table and the internal position within that accessory.
2. System checks if *extpos* and *intpos* are valid positions; if not, then XYF1
3. System checks if it is possible to move over the highest accessory; if not, then XYF2
4. If the dispense head is already at the external position *extpos*, then the operation *ZMove* is called to move the head to the top of the current accessory leaving a safety margin between the head and the accessory.

5. If the dispense head is not at the external position *extpos* and it is not safely above the accessories on the table, the operation *ZMove* is called to move the head to the height of the highest accessory with a safety margin.
6. System calculates parameters for *XMove* and *YMove* from the parameters *extpos* and *intpos*
7. System moves to the desired position by calling the operations *XMove* and *YMove*.
8. The system signals success of moving.

Failure reports:

XYF1. Input parameters *extpos* and *intpos* do not represent a valid position.
Remedy: User changes parameters *extpos* and *intpos*, identifies the cause and resumes or aborts the calling protocol execution.

XYF2. Accessory is too high.
Remedy: User removes the accessory or aborts the calling protocol execution.

The typical course of events for the vertical movement is similar and can be found on the Formal Fillwell homepage [FormFill03].

The abstract specification of the horizontal movement of the component XYZ-Driver is given in the statechart diagram in Appendix B.1. In order to move horizontally XYZ-Driver must first move high enough not to collide with the accessories on the operating table. XYZ-Driver receives the command to move modelled with the *XYMoveCommand* operation. The parameter *p_extpos* is the position of the plate to move to, *p_intpos* is the internal position in that plate and the other three parameters are the speeds to move with in the different directions. Upon this command XYZ-Driver enters state *prep_xyzmove* to indicate that it is ready to move. If all parameters are valid and the yellow lamp is on, the operation *XYMove* is enabled and the moving in the vertical direction is performed by calling the operation *ZMove* in Z-Driver. If the z-movement was successful the operation *ZMoveOk* is enabled and performs the horizontal moving by calling *XMove* and *YMove* in parallel. The operation *XYMoveOk* is enabled if also the horizontal moving was successful and returns the XYZ-Driver to state *idle*. If anything goes wrong while moving, a failure operation will be enabled instead that take XYZ-Driver to a suspended state.

The vertical movement is shown as a statechart diagram in Figure 9. First XYZ-Driver receives a command to move with *ZZMoveCommand* operation, were the parameter *p_zrefpoint* is the reference point, *p_offs* is the offset from that reference point and *p_zspeed* is the speed to move with. The component XYZ-Driver then enters the state *prep_xyzmove* to indicate that it is ready to move. The operation *ZZMove* is enabled, if the yellow lamp is on and the parameters are valid. It moves by calling operation *ZMove*. The operation *ZZMoveOk* is then enabled when the move was successful and returns XYZ-Driver to state *idle*. If something goes wrong while moving, XYZ-Driver enters state *xyz_msuspended*. From the suspended state there is either a remedy operation taking XYZ-Driver back to its previous state or an operation aborting the execution.

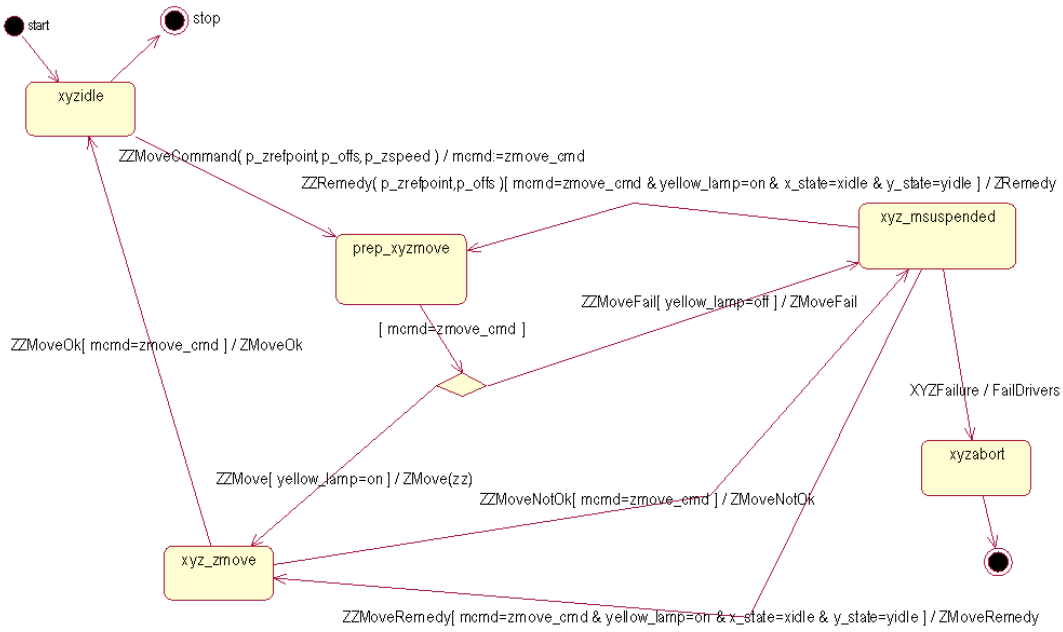


Figure 9: Statechart diagram for the vertical movement of XYZ-Driver.

4.4.2 Feature 1: Coarse grained positioning

In the first refinement step we introduce the positions of the plates on the operating table, *extpos*. For the actual movement of the dispense head over the table this external position is transformed to x- and y-coordinates, *XCoord* and *YCoord*, which give the exact position in micrometers. The variables *XDest_Coord* and *YDest_Coord* are introduced to store the desired position to move to. The vertical positioning of the dispense head is given with a coarse granularity in this step, *z_refpoint*. The three reference points for the head are: (1) the height of the plate, (2) the liquid level, and (3) the bottom level of the plate. The reference point is transformed to a z-coordinate, *ZCoord*, for the vertical movement. XYZ-Driver reads the heights and depths of the accessories on the operating table.

The z-coordinate should always be above the highest accessory of the operating table when, moving from one plate to another:

xyz_state1 □ {xyz_zmove1,xyz_xymove1} /\ mcmd=xymove_cmd /\ ¬(extpos=old_extpos)
□ zmid+s_margin □ ZDest_Coord

If the head is only moving within one external position on the table, the head only has to be above the plate in that particular position. Furthermore, the tips should never hit the bottom of the plate.

ZCoord ∈ zminAllowed(old_extpos)..zmax

The operations in the components are refined to take into account the new features. For example, the operations in the component X-Driver take into account the variables *XCoord* and *XDest_Coord* as shown in Figure 10. When the operation *XMove* is invoked the yellow lamp should be on. The operation *XMoveOk* is enabled if the current position *XCoord* of the dispense head is the desired position *XDest_Coord*. The suspended state *xsuspended* has been divided into two states *xsuspended1* and *x_msuspended1* to keep track of where an error has occurred. If the moving fails, the state of X-Driver will be *x_msuspended1*. Other failures will take X-Driver to state *xsuspended1*. The statechart diagrams for Y- and Z-Driver are similar and are not shown here.

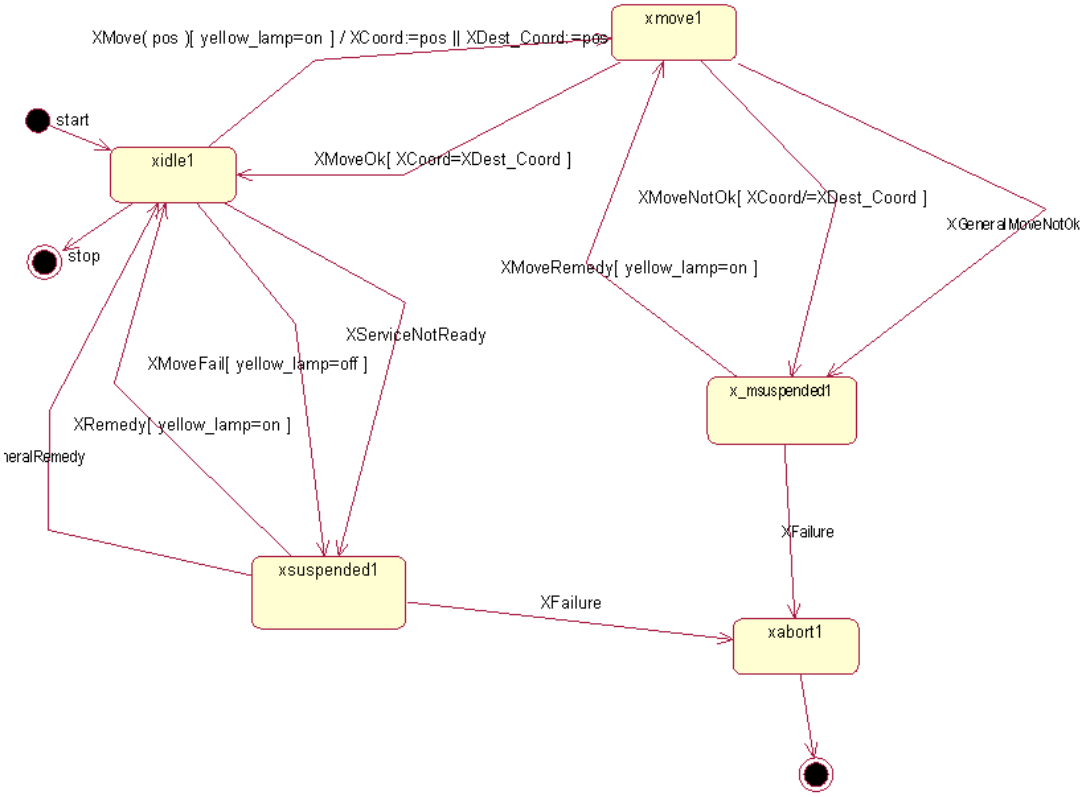


Figure 10: Statechart diagram of the refined X-Driver.

In XYZ-Driver the guard of the operation *XYMove* is strengthened in such a way that the operation is enabled if the desired position is not the current position and the dispense head is not high enough, i.e., above the highest accessory on the operating table, but it is possible to move it high enough.

¬(extpos=old_extpos) ∧ zmid+s_margin>ZCoord ∧ zmid+s_margin<zmax

Furthermore, the operation is enabled if the dispense head is already in the desired position and it is possible to move the dispense head above the accessory in that position.

extpos=old_extpos /\ zmid+s_margin<zmax

If the dispense head is already above the highest accessory, there is no need to move vertically.

When the dispense head is high enough to be moved horizontally, the operation *XYMoveZOk* is enabled and moves the head to the desired x- and y-coordinates. Finally, if the correct position was reached, the operation *XYMoveOk* is enabled. Should any of these moving operations fail, the component XYZ-Driver becomes suspended. In this refinement step the suspended state for XYZ-Driver has been divided into three states, *xyz_msuspended1*, *xyz_zsuspended1* and *xyz_xysuspended1*, to correspond to the respective originating states, *prep_xyzmove* (XYZ-Driver prepares to move), *xyz_zmove* (XYZ-Driver has moved vertically) and *xyz_xymove* (XYZ-Driver has also moved horizontally).

The statechart diagram for the horizontal movement of XYZ-Driver is fiven in Appendix B.2. The complete specification of the first refinement step can be found on the Formal Fillwell homepage [FormFill03].

4.4.3 Feature 2: Detailed vertical positioning

In the second refinement step we introduce an offset variable, *offs*. The offset indicates how much higher or lower than the given reference point the dispense head should be moved. The reference point and the offset together give the exact z-coordinate to move to.

Since the actual height of accessories on the operating table is introduced, the invariant guarantees that XYZ-Driver is high enough when moving horizontally within a plate.

xyz_state1 ∈ {xyz_zmove1,xyz_xymove1} /\ mcmd=xymove_cmd /\ extpos=old_extpos
⇒ AccessoryHeight(old_extpos)+s_margin ≤ ZDest_Coord

The components X-, Y- and Z-Driver are not changed in this refinement step. The service *XYMove* to move horizontally is also the same as in the previous step. However, the service *ZZMove* to move vertically is refined to take into account the exact z-coordinate *ZCoord*. The move can fail if the z-coordinate is not within range. The specification of this refinement step can be found on the Formal Fillwell homepage [FormFill03].

4.4.4 Feature 3: Detailed horizontal positioning

Finally, we add internal positions, *intpos*, within a plate, as well as the speed for moving in the three directions, *xspeed*, *yspeed* and *zspeed*. The internal positions of a plate depends on the number of wells in the plate, as well as on the number of tips attached to

the dispense head as explained in subsection 4.3.4. The exact x- and y-coordinates to move to are calculated from the given external and internal position.

In this refinement step the speed is taken into account in the components X-, Y- and Z-Driver. The operations in component XYZ-Driver are refined to take into account the internal positions of the plates. The internal positions should be in range concerning the number of wells in the current plate and the number of tips in the dispense head as described in Table 1. The statechart diagram of service *XYMove* is given in Appendix B.3.

4.4.5 Control system development

As a final step the component XYZ-Driver is partitioned into plant and controller. The variables modelling the x-, y- and z-coordinates are the sensors of XYZ-Driver and the rest of the variables are considered to be actuators.

4.4.6 Proving correctness

The proof obligations described in subsection 2.4.3 are generated for each step. In Table 3 we show the number of B machines, lines of code and proof obligations for the development of XYZ-Driver.

XYZ-Driver	Machines	Lines	Obvious p.o.	Generated p.o	Autoproved	Percent
spec	**7**	**790**	**241**	**440**	**415**	**94 %**
ref. step 1	**10**	**1 300**	**1 433**	**2 164**	**2 085**	**96 %**
ref. step 2	**14**	**1 670**	**1 027**	**160**	**151**	**94 %**
ref. step 3	**14**	**2 240**	**3 214**	**802**	**769**	**96 %**

Table 3: Quantitative information of the XYZ-Driver development.

A large number of proof obligations are generated for the first refinement step where the coarse grained positions are introduced. Most of these proof obligations are, though, automatically discharged. We can note that the rate of the automatically discharged proof obligations is at a satisfactory level for all the refinement steps with 94% as the lowest rate. The proofs that were not discharged automatically have been proved interactively. No new user defined rules had to be added during the interactive proof. Merely proof tactics were needed for the interactively discharged proof obligations. Since both Dispenser and XYZ-Driver use the component Operating table, the information about the machines Operating table is included in the figures presented here in Table 3, as well as in Table 2.

4.5 The component Protocol runner

In order to perform sample testing with the Fillwell workstation the user provides commands to the workstation via a protocol. The component Protocol runner loads the user protocol, reads the commands in the protocol and coordinates the components XYZ-Driver and Dispenser according to these commands.

4.5.1 The specification

The component Protocol runner provides five services to the user; *LoadProtocol* for loading a new protocol into the Fillwell, *ReadProtocol* for reading and executing a protocol, *Pause* for pausing the execution of the protocol, *Continue* for continuing after the pause, and *Emergency* for terminating execution of any activity of Protocol runner. The protocol consists of three sections; the first section introduces the constants, the second verifies the configuration of the accessories on the plate, and the third contains the actual protocol commands. The protocol commands concern moving the dispense head, setting the speed of the movement, as well as aspirating and dispensing liquid and air. Additionally, there is a command for washing the tips of the dispense head. The protocol sections and their commands are given in Appendix C.1.

When Protocol runner reads and executes the commands in the protocol it uses the services of the components Dispenser and XYZ-Driver. Dispenser and XYZ-Driver in turn use Operating table for data accessing. In order to perform the washing of tips we introduce a component Tipwasher that contains the two pumps of the tip washer accessory. One of the pumps fills the tip washer accessory with washing liquid, while the other pump empties it. Tipwasher provides the services to set on and off the two pumps. The component diagram that describes the relations between all these components is shown in Figure 11.

Each service of the component Protocol runner is initiated by the user. When the service ReadProtocol is requested, Protocol runner reads the protocol line by line and executes the command on the current line. If an error occurs in the protocol, an error report is given, the execution of the protocol aborts and a new protocol should be loaded. The typical course of events for the service ReadProtocol is given in Appendix C.2.

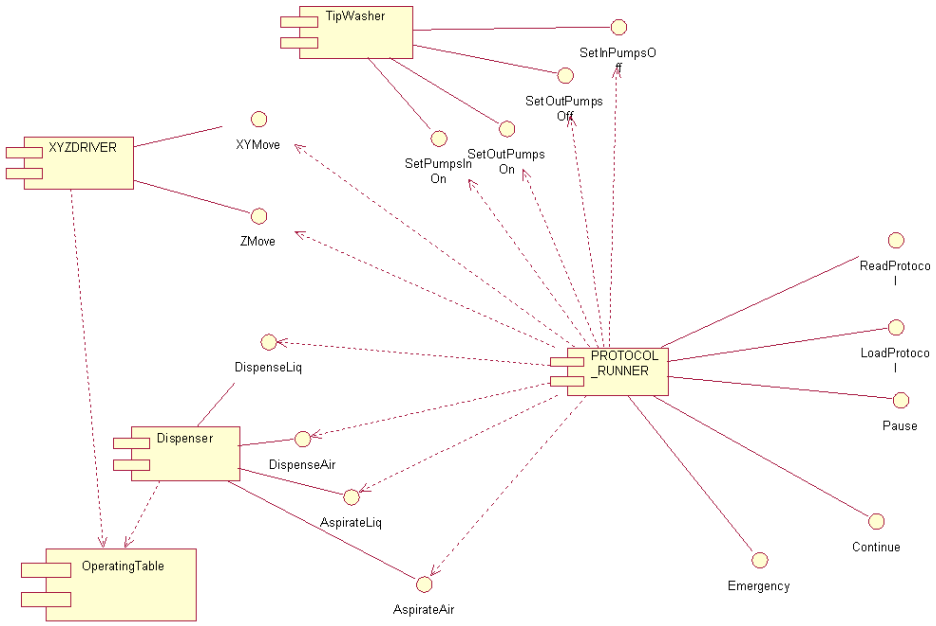


Figure 11: Component diagram for the Fillwell system.

The Fillwell workstation has to fulfill certain safety requirements. When the commands of the protocol are executed, Protocol runner coordinates the components Dispenser and XYZ-Driver in such a way that these conditions are satisfied. Hence, the invariant of Protocol runner states that the gantry is not allowed to move the dispense head while it aspirates or dispenses. Moreover, dispense head is not allowed to aspirate or dispense (it has to be idle or suspended), when the gantry is moving the head.

$\neg$ (dstate $\in$ {didle, didle_susp, dabort}) $\Rightarrow$ xyz_state $\in$ {xyzidle, xyzsuspended, xyzabort}

The workstation has a yellow lamp, *yellow_lamp*, indicating when the dispense head is allowed to move. The yellow lamp should always be switched on when Protocol runner is executing a command to move (*p_state=pworking*), has read an erroneous parameter of a command (*p_state=param_susp*) or has been requested to pause (*p_state=ppause*) and has not yet halted (*pause_state≠ pause_home_all*).

(p_state = pworking $\Rightarrow$ yellow_lamp = on) $\wedge$ (p_state = param_susp $\Rightarrow$ yellow_lamp = on)
$\wedge$ (p_state = ppause $\wedge \neg$ (pause_state = pause_home_all) $\Rightarrow$ yellow_lamp = on)

The component XYZ-Driver is not allowed to move, nor is Dispenser allowed to dispense or aspirate, when Protocol runner is waiting for a new protocol (*p_state=pinit*) or pausing (*pause_state=pause_home_all*). Additionally, the yellow lamp should be off when Protocol runner is paused.

(p_state = pinit $\Rightarrow$
xyz_state $\in$ {xyzidle,xyzsuspended, xyzabort} $\wedge$ dstate $\in$ {didle, didle_susp, dabort})
$\wedge$ (p_state = ppause $\wedge$ pause_state = pause_home_all $\Rightarrow$

xyz_state ∈ {xyzidle,xyzsuspended,xyzabort} ∧ dstate ∈ {didle,didle_susp,dabort}
∧ yellow_lamp = off)

The behavior of the component Protocol runner is described with statechart diagrams. The statechart diagrams for Protocol runner in Appendix C are simplified to become more readable and do not have a one to one correspondence to the B machines in the Formal Fillwell development [FormFill03].

The statechart diagram in Appendix C.4 describes how external events affect Protocol runner. The operation *LoadProtocol* loads a new protocol and changes the state of Protocol runner to *pworking*. This state indicates that Protocol runner is decoding and executing the protocol. The operation *EndProtocol* is enabled when the command *End of protocol* has been read and it changes the state of Protocol runner back to idle.

Let us take a closer look at the state *pworking* for decoding and executing the protocol. The statechart diagram describing this state is given in Appendix C.5. The initial state of the diagram is *cmd_not_ready* indicating that the command has not been dealt with yet. The operation *ReadCommand* reads the command on the current protocol line. The operation *LineReady* is enabled when the end of the current protocol line is reached and then changes the working state to *line_ready*. When Protocol runner is in state *line_ready* it means that the reading of a protocol line has been completed and the command read from the protocol can be decoded and executed.

The command read from the protocol determines which operation is enabled. Command *VerifyConfig* for verifying configuration enables operation *PverifyConfig*, commands *XX, YY, ZZ* and *none* for adding constants enable *AddConstant*, command *Loop* for starting a loop enables *LoopInProtocol*, *Begin* for beginning a new section in the protocol and *End* for ending a section or a loop in the protocol enable *ControlProtocol* and commands *SetXSpeed, SetYSpeed* as well as *SetZSpeed* for setting the speed enable *DecodeCommand2*. The operations change the state of Protocol runner to *cmd_not_ready* indicating that it is ready to read a new command and Protocol runner becomes ready to read the next protocol line. The non-administrative commands enable the operation *DecodeCommand1* and the state of Protocol runner is changed to *cmd_ready* indicating that the protocol command is ready to be executed.

When Protocol runner has read a non-administrative command it is ready for co-operation with XYZ-Driver and Dispenser. If the protocol command is *GotoAccessoryPosition* or *ZGotoAccessory* for moving horizontally and vertically, then *MoveCommand* is enabled. The commands *Aspirate* or *AspiratePrimaryAir* for aspirating liquid or air enable *AspirateCommand* while *Dispense* or *BlowPrimaryAir* for dispensing liquid or air, enable operation *DispenseCommand*. If the number of parameters and their types are valid the command and the parameters are passed on to XYZ-Driver or Dispenser. The state of Protocol runner is then changed to *cmd_not_ready* and Protocol runner becomes ready to read the next line in the protocol. Note that the operations above only start the execution of the command. The operations of XYZ-Driver and Dispenser

are executed in parallel with the operations of Protocol runner to complete the execution of the command.

When running a protocol the workstation has to dispense and aspirate different kinds of liquid. In order to keep the samples clean, the workstation also has to wash the tips. If the command is *WashTips* then the operation *WashTipsCommand* is enabled and the state of Protocol runner becomes *wash*, starting the washing of the tips. When the washing is finished the operation *FinishWashing* changes the state of Protocol runner to *cmd_not_ready* and a new protocol line may be read. The typical course of events for the washing procedure including possible failures is given in Appendix C.3.

When an error occurs during the protocol reading, Protocol runner is suspended. Protocol runner has two suspended states, *param_susp* and *psuspended*. The state *param_susp* means that an error in the parameters of the protocol has been detected. The state *psuspended* indicates that a serious error has occurred when executing the protocol or that one of the components XYZ-Driver or Dispenser is malfunctioning. A remedy operation in Protocol runner takes it to the state *init*. This models that the execution of the protocol is stopped if an error occurs in the protocol. When the components XYZ-Driver or Dispenser are malfunctioning, they enter the state *abort*. This error situation is detected by Protocol runner.

The Fillwell workstation has a pause button and an emergency button that the user can press when he/she wants to stop the execution of the system. When Protocol runner is in state *pworking* it can be paused. The operation *PauseCommand* changes the state of Protocol runner to *ppause*. This command enables a number of operations that moves the dispense head up and then to its home position. When the home position is reached the yellow lamp is switched off. When the pause button is pressed once, the Fillwell workstation is paused and the dispense head is moved to its home position. By pressing this pause-button once again the user allows the system to continue with the protocol from where it was interrupted. The operation *ContinueCommand* switches on the yellow lamp and calls XYZ-Driver to move the dispense head horizontally to the old accessory. If the last command before the pause command was a vertical move Protocol runner calls XYZ-Driver to move the dispense head to the height it was at before the pause command was issued. Finally, the state of Protocol runner is changed to the state it was in when it was paused.

In case of an emergency the user can press the emergency button on the Fillwell workstation and the system is stopped immediately. A red lamp indicates that the emergency button has been pressed. The operation *EmergencyCommand* changes the state of Protocol runner to *pabort*. Furthermore, it calls the emergency operations in the components Dispenser and XYZ-Driver to stop them as well and change their states to abort.

4.5.2 Feature 1: More detailed data

In the first refinement step we introduce the variable *param_list*. It contains parameters from the current protocol line. The parameters are of type integer values. A variable *param_no* is introduced for keeping track of the number of parameters read from the protocol. For example when the command in the protocol is *GotoAccessoryPosition* the variable *param_list* contains the value of the external position to move to, *extpos,* at index 0 and the value of the internal position, *intpos,* at index 1. For command *GotoAccessoryPostition* the variable *param_no* has the value 2.

In this step we also introduce more features for the washing procedure. The amount of liquid and air to aspirate and dispense during the washing, as well as the number of times that the tips should be washed are explicitly given for the command *WashTips*.

When the Protocol runner has read a command from the protocol, it calls the requested operations in XYZ-Driver or Dispenser to initiate the execution of the command. In order to model that the component might not always be ready to commit immediately we introduce the variables *xyz_waited* and *d_waited*. When the requested component XYZ-Driver or Dispenser is busy *xyz_waited* or *d_waited*, respectively, is set to *TRUE* to model that Protocol runner waits for a while for the component to become idle. The variable is set to *FALSE* when the component becomes ready to commit or after time out for waiting.

The statechart diagram for the refined Protocol runner is given in Appendix C.6. In this step the operations *XYZWait* and *DispWait* and more accurate failure operations are introduced. Protocol runner can wait for the components XYZ-Driver and Dispenser to become ready when it is in state *pworking* or *ppause*, i.e., it is executing the protocol or pausing. The operations *XYZWait* and *DispWait* set the boolean variables *xyz_waited* or *d_waited* to *TRUE* when XYZ-Driver or Dispenser are busy. If XYZ-Driver or Dispenser is not ready after the waiting period the operations *DispWaitFail* or *XYZWaitFail* change the state of Protocol runner to *psuspended*.

In this refinement step the reading of the commands is extended with a more detailed treatment of the parameters. The parameters can be constant names, array constants or ordinary integer values. The new operations *ConstantFound, ArrayConstantFound* and *ReadParameter* are enabled when the parameter is a constant name, a constant array or an integer value, respectively. The operations translate the parameters to ordinary integers and add the parameters to the variable *param_list*. The statechart diagram of the protocol reading and decoding is given on the Formal Fillwell homepage [FormFill03].

The opertion *TipWashing* is also refined in this step. The statechart diagram in Appendix C.7 gives a graphical view of the washing process. When the command *WashTips* is executed the dispense head is first moved above the tipwasher accessory. The starting point in the statechart diagram in Appendix C.7 is the state *xymoved* where the dispense head is already above the tipwasher. The operation *WashTipsZMove* is enabled if there is

old liquid in the tips and prepares to remove the liquid. The actual washing starts from state *asp_air*. The tips are washed in *cycle_count* number of washes. The pumps fill and empty the tipwasher accessory in each cycle. If all cycles are not finished (*cycle_count>0*), operation *WashTipsFinishCycle* starts a new cycle, else it finishes the washing. The tips are then washed by aspirating and dispensing washing liquid *wash_count* number of times via the operations *WashTipsAspirate* and *WashTipsDispense*. If the tips should be further washed (*cycle_count>0*), the state is changed to *asp_air* again. On the other hand, if the washing is ready (*cycle_count=0*), Protocol runner dispenses primary air with operation *WashTipsDispense* and moves up above the tipwasher with operation *WashTipsZMove* changing the state to *finish*.

The component Protocol runner fills the tipwasher accessory with washing liquid and empties it by calling *SetInPumpOn* and *SetOutPumpOn*, respectively, in component Tipwasher. Tipwasher notifies Protocol runner when it is done by setting the variables *in_pumps_done* and *out_pumps_done* to *TRUE*, respectively.

4.5.3 Feature 2: Protocol Lines

In the second refinement step we introduce a protocol and protocol lines. The variable *protocol* models the protocol with numbered lines. A variable *constant_table* for storing constant names is also introduced. The protocol consists of a constants-section where constants are defined, a configure-section where it is verified that the configuration of accessories expected by the protocol is indeed the same as the actual configuration, and a protocol command-section which contains the actual commands for moving, dispensing and aspirating. The protocol is read, decoded and executed one line at a time. An item in a protocol line is either an integer, a constant name or a loop counter variable (an index variable).

The first item of a line is the command, which is of type constant name. The command is followed by a number of parameters as shown in Figure 12. A protocol line is read one item at a time until end of line is encountered. Integer parameters are read and added to a parameter list which will be used when the command is executed. When parameters in the form of constants occur, their values are looked up in the constant table and added to the parameter list. All the constants in the configuration- and protocol command-section must be defined in the constants-section of the protocol.

Line number	**0 (Command)**	**1 (Param. 1)**	**2 (Param. 2)**	**3 (Param. 3)**	**....**
100	Dispense	200			
101	GotoAccessoryPosition	extpos	intpos		
....	....	....	....	....	

Figure 12: The variable *protocol*.

There are two variables for keeping track of the current position in the protocol, *line_no* and *protocol_str_no*. The variable *line_no* is the current line number and *protocol_str_no* is the current string to read from that line. Let us look at the protocol in Figure 12. On line number 100 the command to be executed is *Dispense* and the liquid amount to dispense is 200. On line 101 the command is *GotoAccessoryPosition*, the external position is given as constant *extpos* and the internal position as constant *intpos*. Since *extpos* and *intpos* are constants, their values have to be looked up in the constant table *constant_table* shown in Figure 13, where constant *extpos* has value 5 and *intpos* has value 12. Hence, the command says that the dispense head should be moved to internal position 12 in accessory 5.

Constant name	**Value**
extpos	5
intpos	12
...	...

Figure 13: The variable *constant_table*.

The operations of Protocol runner are refined to deal with the detailed protocol. It is mainly the operations that take care of reading and executing the protocol that are changed in this step. They are depicted in three statechart diagrams of which two are given in Appendix C.8 (for parsing a line) and C.9 (for decoding the command and the parameters) and one (for executing commands) is given merely on the Formal Fillwell homepage [FormFill03]. For readability these diagrams are slightly simplified versions of the corresponding B specifications.

The statechart diagram in Appendix C.8 describing line parsing starts in state *cmd_not_ready* and ends in state *line_ready*. Operation *ReadCommand* is enabled when the length of a protocol line is greater than zero and the type of the first element on the protocol line is a command. It then reads the command from the protocol. Operation *ReadParameter* is enabled when the end of protocol line has not yet been reached and the protocol line is not too long. Furthermore, the parameter should not be a constant name, but an integer value:

protocol_str_no < size(protocol(line_no)) ∧ protocol_str_no < max_line_length
∧ param_no < max_line_length ∧ ¬ (cmd ∈ {XX,YY,ZZ,PP,Loop,none} ∧ param_no = 0)
∧ (protocol(line_no)(protocol_str_no)) ∈ {vv}*MSTRING.

The operation *ReadParameter* then adds the parameter to the parameter list. On the other hand, if the parameter is a constant from the constant table and the next item is not a loop counter variable (an index variable) the operation *ConstantFound* is enabled:

protocol_str_no < size(protocol(line_no)) ∧ protocol_str_no < max_line_length
∧ param_no < max_line_length ∧ ¬ (cmd ∈ {XX,YY,ZZ,PP,Loop,none} ∧ param_no = 0)
∧ (protocol(line_no)(protocol_str_no)) ∈ dom(constant_table)
∧ (protocol(line_no)(protocol_str_no)) ∈ {cc}*MSTRING
∧ ¬ ((protocol(line_no)(protocol_str_no+1)) ∈ {cl}*MSTRING).

It looks up the constant name in the constant table and adds the corresponding parameter value to the parameter list. The operation *ArrayConstantFound* is enabled when the parameter is a constant from the constant table followed by an index variable and it also adds the parameter to the parameter list. If a new constant name, which is not in the constant table, is encountered and the command is *XX*, *YY*, *ZZ*, *PP* or *none*, the operation *ConstantNameFound* is enabled and adds the constant name to the temporal variable *t_constantname*. The operation *LineReady* is enabled when the end of line has been reached and the protocol is not too long.

When the complete protocol line has been read, Protocol runner decodes the line. The statechart diagram for decoding the protocol line is given in Appendix C.9. A variable *control_state* is introduced to keep track of whether Protocol runner is currently reading constants, checking configuration or executing protocol commands. The operation *ControlProtocol* changes the variable *control_state* to keep track of the clauses in the protocol upon the commands *Begin* and *End*. The operation *AddConstant* adds constants to the constant table when the command is *XX*, *YY*, *ZZ*, *PP* or *none* in the constants-section of the protocol, the constant name is of correct type and the parameter list contains the constant value:

cmd ∈ {XX,YY,ZZ,none} ∧ control_state = p_const_start
∧ t_constname ∈ {cc}*MSTRING ∧ 0 ∈ dom(param_list).

The operation *PVerifyConfig* verifies the configuration of the accessories, when the configuration-section of the protocol is being read. The operation *StartLoopInProtocol* starts a new loop upon command *Loop* within the protocol-section. The operation *EndProtocol* is enabled when there are no open loops and the command is *End.* All these commands for controlling the execution of the protocol changes the state of Protocol runner to *cmd_not_ready* indicating that Protocol runner is ready to read a new command.

Upon the commands operating the workstation by moving, aspirating, dispensing and setting the speed within the protocol command-section, the operations *DecodeCommand1* and *DecodeCommand2* are enabled. They prepare the execution of the command by changing the state of Protocol runner to *cmd_ready*. The actual execution of the commands in the protocol command-section is performed via the components Dispenser, XYZ-Driver and Tipwasher. Depending on the command an operation is enabled calling the corresponding operation in one of the components.

4.5.4 Feature 3: Control structures

In the third refinement step nested loops and array constants are introduced. For the nested loops we keep track of the return address and the loop counters with the help of stacks. The variable *stack_loop_ret_addr* is a stack that contains return addresses for the loops, *stack_loop_counter* contains names of the loop counters and *stack_loop_ntimes*

contains the number of times to loop. The variable *stack_pointer* is a pointer to the top element of the stacks. The top element of all three stacks always refer to the same loop.

Array constants are constants that contain a list of values that can be indexed by a loop counter. The constant *intpos* in Figure 14 is an array constant that has three values: 1, 2 and 3. When reading an array constant in the protocol command-section the next item on the line must be a loop counter. The name of this loop counter need to be in the stack *stack_loop_counter*, i.e., the loop counter has to be in use. The value of the loop counter is looked up in the constant table to get an index value. The value of the array constant at that index is then added to the parameter list. For example, if the value of the loop counter is 2, the value added to the parameter list for *intpos* is 3 in Figure 14.

Constant name	**0 (index 0)**	**1 (index 1)**	**2 (index 2)**	**....**
intpos	1	2	3	
extpos	5			
....	....	...	....	....

Figure 14: The variable *constant_table1* variable.

The variable *constant_table* is refined by the variable *constant_table1* in this step to take into account also array constants. The values at index 0 in *constant_table1* corresponds to the values in the old variable *constant_table*.

(∀xx.(xx ∈ dom(constant_table) ⇒ constant_table(xx) = constant_table1(xx |->0)))

This is shown in Figure 14 where constant *extpos* is an single value constant with value 5 at index 0.

The process of adding an array constant to the constant table is depicted in Figure 15. In the example in Figure 15 the current line is 10 in the protocol (1). The line contains a command *none* to add a constant with the array constant name *intpos* and its values 1 and 2. The constant name is stored in the variable *t_constname* (2) while its values are stored in the parameter list (3). The constant name is then added to the constant table *constant_table1* (4). Finally, the whole list of values from the parameter list is added to the constant table for that constant (5). This stepwise addition of an array constant to the constant table is due to the fact that the feature of reading a parameter of type constant is superposed on the general parameter reading.

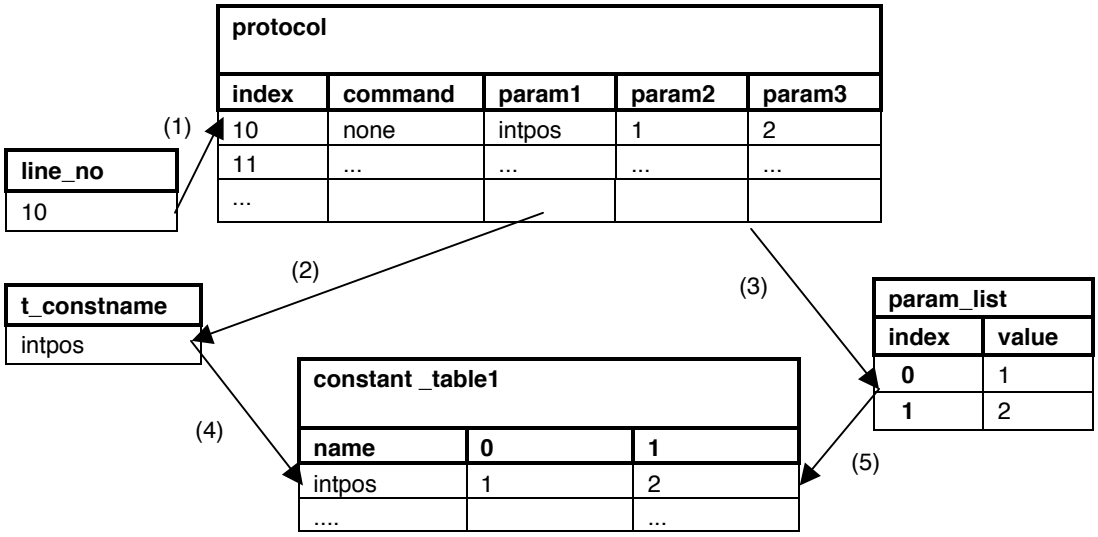


Figure 15: The process of adding an array constant to the constant table.

The refined statechart diagram describing the decoding is given in Appendix C.10. The operation *StartLoopInProtocol* is enabled when the command is *Loop*, the protocol-section in the protocol is being read and the parameter giving the number of times to loop is greater or equal to zero. Furthermore, the loop counter should be of type index variable and there should not be a counter with the same name in use:

cmd = Loop ∧ control_state = p_protocol_start ∧ 0 ∈ param_list(0)
∧ t_constname ∈ {cl}*MSTRING ∧ ¬ (t_constname ∈ dom(dom(constant_table1))).

If the *Loop* command is encountered inside another loop, i.e., *stack_pointer* is greater than zero, then the operation *StartLoopInProtocol* also requires that there are not too many loops, that the previous return address is smaller than the new return address *line_no+1* and that the loop variable is not already in use:

0 < stack_pointer < sp_max ∧ stack_loop_ret_addr(stack_pointer) ∈ line_no
∧ ¬ (t_constname ∈ stack_loop_counter[1..stack_pointer]).

In this way *StartLoopInProtocol* guarantees that the loops are properly nested when it starts a new loop.

The operation *LoopInProtocol* is enabled when the command is *End* and Protocol runner is inside a loop and the value of the loop counter is less than the number of times to loop. The action will then change the current line *line_no* to the line number stored on top of the return address stack and increase the value of the loop counter in the constant table *constant_table1*. The operation *EndLoopInProtocol* is enabled when Protocol runner is executing a loop and the value of the loop counter is equal to the number of times to loop. The operation ends the loop by decreasing the value of *stack_pointer*. The operation

EndProtocol is enabled when the command is *End* and the stacks are empty (*stack_pointer=0*), i.e., when Protocol runner is not executing a loop. This operation ends the execution of the protocol by switching the yellow lamp off and taking Protocol runner to its initial state *pinit*.

4.5.5 Proving correctness

The component Protocol runner has been developed and proved using Atelier B in the same way as Dispenser and XYZ-Driver. The lines of code for each development step as well as the number of proof obligations generated and proved are given in Table 4. In the table we only refer to the machines that are not included in the components XYZ-Driver or Dispenser.

Protocol runner	Machines	Lines	Obvious p.o.	Generated p.o	Autoproved	Percent
spec	**4**	**710**	**1 418**	**617**	**572**	**93 %**
ref. step 1	**4**	**880**	**9 654**	**289**	**289**	**100 %**
ref. step 2	**4**	**1 120**	**9 262**	**293**	**280**	**96 %**
ref. step 3	**4**	**1 200**	**12 457**	**243**	**198**	**81 %**

Table 4: Quantitative information on Protocol runner development.

The percentage of the automatically proved proof obligations is high for the two first refinement steps (≥ 93%), but low for the third one (81%). The low percentage is due to the use of quantified expressions, as well as complex function definitions for the array constants in the invariant. The proof obligations that were not automatically discharged by Atelier B were, however, discharged with the interprover. Even if up to 19% of the proof obligations were proven interactively no new user defined rules needed to be introduced during these interactive proofs. The prover only required help with tactics to discharge these proof obligations.

5. Conclusions

The healthcare case study presented in this paper deals with the development of a drug discovery system, Fillwell. It has been carried out in co-operation with Wallac within the MATISSE-project [MATISSE03] using an industry-as-laboratory approach. The goal of the case study was to make the industrial partner Wallac aware of the benefits of using formal methods. We have developed the Fillwell system formally at Åbo Akademi University, guided by the experts at Wallac on the system requirements. An informal development of the system has simultaneously been performed at Wallac.

In the healthcare case study the formal development was split up in the same components as the informal one: Dispenser, XYZ-Driver and Protocol runner. The component Protocol runner coordinating the execution of Dispenser and XYZ-Driver. The methodology used in the case study is a combination of methods, UML, Action Systems, the B Method and safety analysis. This methodology has been developed during many years at Åbo Akademi University. The components of the Fillwell system were formally expressed as B Action Systems via UML-diagrams. The system was developed in a stepwise manner adding new features in each step and proving their correctness. The documentation was created stepwise via the UML-diagrams during the development.

Due to the complexity of the system, the proof effort was tedious in the formal development. More than half the time required for the developmet was spent on proving the correctness of the refinement steps. During the proving process most of the generated proof obligations were discharged by the automatic prover of Atelier B. However, many proof obligations still needed to be proved with the interactive prover. By minimizing the number and complexity of the proof obligations generated for each refinement step using suitable data structures and expressions the interactive proving effort can be reduced. During the development the automatic prover should be used to discharge the proof obligations, while the interactive prover should be used to inspect the proof obligations that were not proved automatically [MATISSE03]. Only at the end of the development the interactive prover should be used to discharge these proof obligations. When developing safety critical systems it is also important to remember that we can only prove that a system is correct according to its specification. If the specification is wrong, the system will not work correctly. Errors discovered late in the development process often cause wide-ranging changes. Hence, one should be very careful when creating the specification from the requirements of the system.

The case study was qualitatively validated by the development teams at Wallac and Åbo Akademi University. As a conclusion of this validation we can state that formal methods help in the inspection of the code when applied with a graphical interface. Since the formal development was performed at the university no physical product resulted from that development. Hence, we cannot directly state whether formal methods have a positive effect on the final product. We can, however, state that the formal methods as a combination of UML and B had a positive effect on the design process. The typical course of events (use cases) treated the error situations in a very exact way, which was found to be very useful for the development. The B Method forces the developers to specify the system in an unambiguous way and, hence, to think carefully about the precondition of each service and which postconditions these services should establish. In this way the formal development also indirectly had a small positive effect on the informally developed product. The stepwise introduction of features into the system provides a structured way of managing the complexity of the system. In its turn, it facilitates better understanding of the system and, hence, leads to better design decisions.

Finally, we can state that it was a positive experience to apply the methodology integrating UML, action systems, B and safety analysis on a large industrial example as

in this case study with such a positive result. The case study provided inspiration for further development of the methodology.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[BK83] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.

[BS96a] R.J.R. Back and K. Sere. From modular systems to action systems. *Software - Concepts and Tools 17*, pp. 26-39, 1996.

[BS96b] R.J.R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing* 8(3):324-346, 1996.

[BRJ99] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BW98] M. Butler and M. Waldén. Parallel programming with the B Method. Chapter 5 in [SS98], pp 183-195.

[ClearSy03] ClearSy. (15.01.2004) *Atelier B* and *Event B*. *http://www.atelierb.societe.com/*

[MATISSE03] *MATISSE Handbook for Correct Systems Construction.* EU-project MATISSE: Methodologies and Technologies for Industrial Strength Systems Engineering, IST-1999-11345, 2003. *http://www.esil.univ-mrs.fr/~spc/matisse/Handbook*

[FormFill03] Formal development of Fillwell. *http://www.abo.fi/~marina.walden/fillwell/Formal_Fillwell_2002.html*

[PE01] PerkinElmer Life Sciences. Fillwell™2002 – Features Guide, 2001. *http:/www.abo.fi/~marina.walden/Fillwell.pdf*

[PRTWJ01] L. Petre, M. Rönkkö, E. Troubitsyna, M. Waldén and M. Jansson. *A Methodology for co-design based on a 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.Waldén, A Healthcare Case Study. In *Proceedings of RCS'02* - International workshop on Refinement of Critical Systems: Methods, Tools and Experience, Grenoble, France, January 2002. *http://www.esil.univ-mrs.fr/~spc/rcs02/rcs02.html*

[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.

[Sek98] 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.

[SW00] K. Sere and M. Waldén. Data Refinement of Remote Procedures. *Formal Aspects of Computing* 12(4): 278 - 297, December 2000.

[SB00] C. Snook and M. Butler. U2B Downloads. *http://www.ecs.soton.ac.uk/~cfs/U2Bdownloads.htm*

[STW03] C. Snook, L. Tsiopoulos and M. Waldén. A case study in requirement analysis of control systems using UML and B . In *Proceedings of RCS'03 - International workshop on Refinement of Critical Systems: Methods, Tools and Experience*, Turku, Finland, June 2003. *http://www.esil.univ-mrs.fr/%7Espc/rcs03/rcs03.html*

[SW02] C. Snook and M.Waldén. 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. *http://www.esil.univ-mrs.fr/~spc/rcs02/rcs02.html*

[Sto96] N. Storey. *Safety-critical computer systems*, Addison-Wesley, 1996.

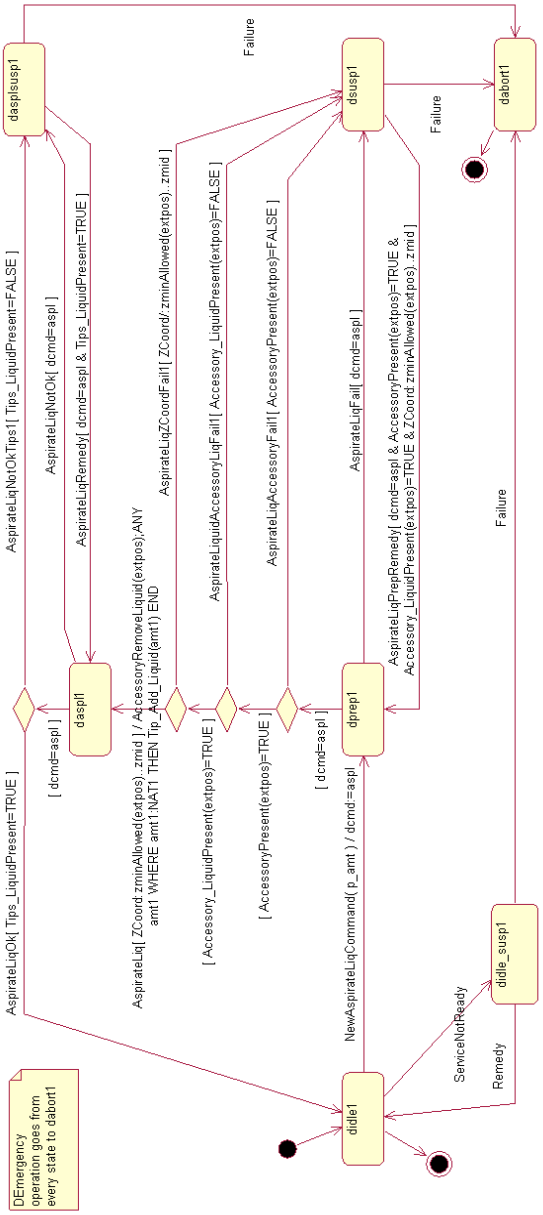
[Tro00] E. Troubitsyna. *Stepwise Development of Dependable Systems*. Turku Centre for Computer Science, TUCS, Ph.D. thesis No.29. June 2000.

[Wal98] M. Waldén. *Distributed load balancing*. Chapter 7 in [SeSe98], pp. 255-300.

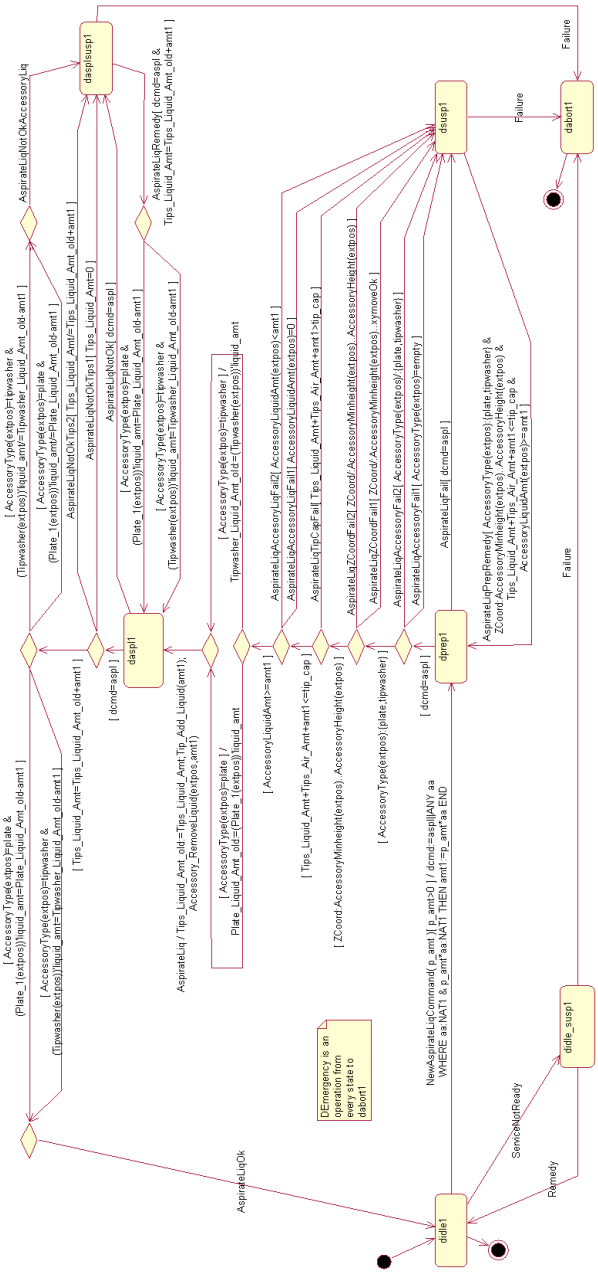
[WS98] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design 13(5-35),* 1998. Kluwer Academic Publishers.

Appendix A

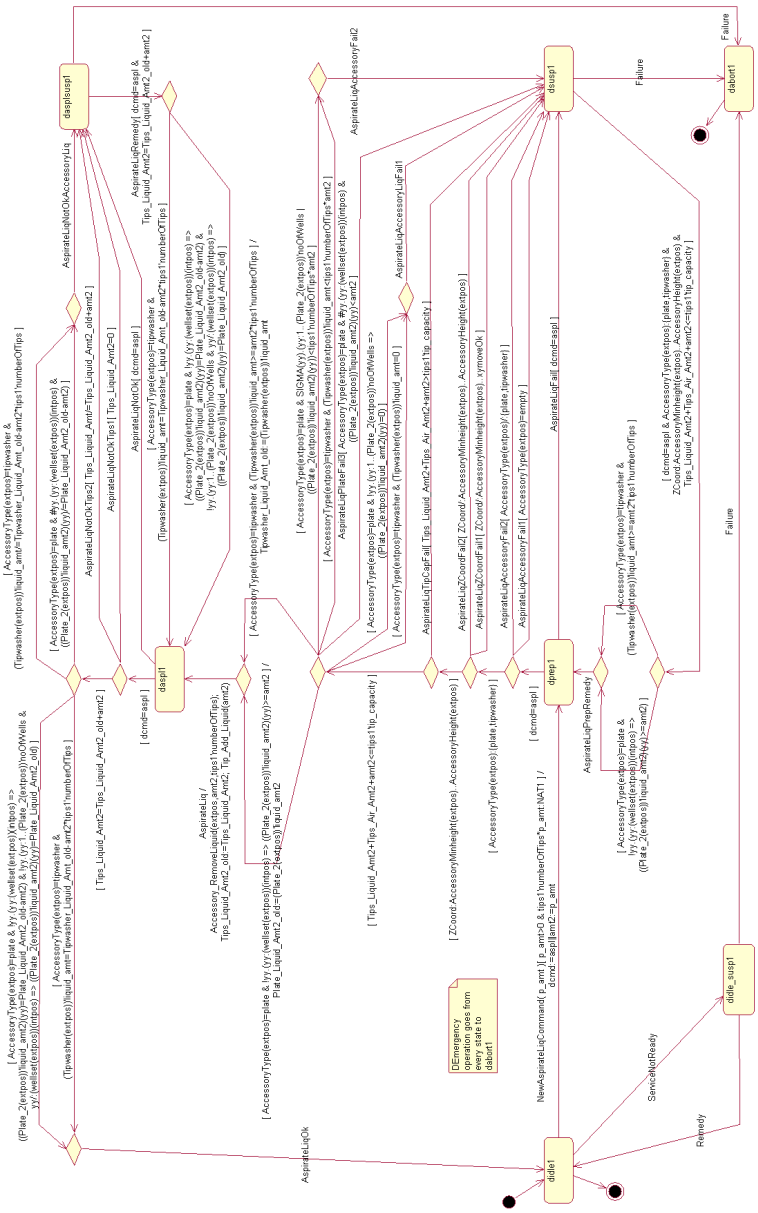
A.1. First refinement step of the operation aspirate liquid in Dispenser.



A.2. Second refinement step of the operation aspirate liquid in Dispenser

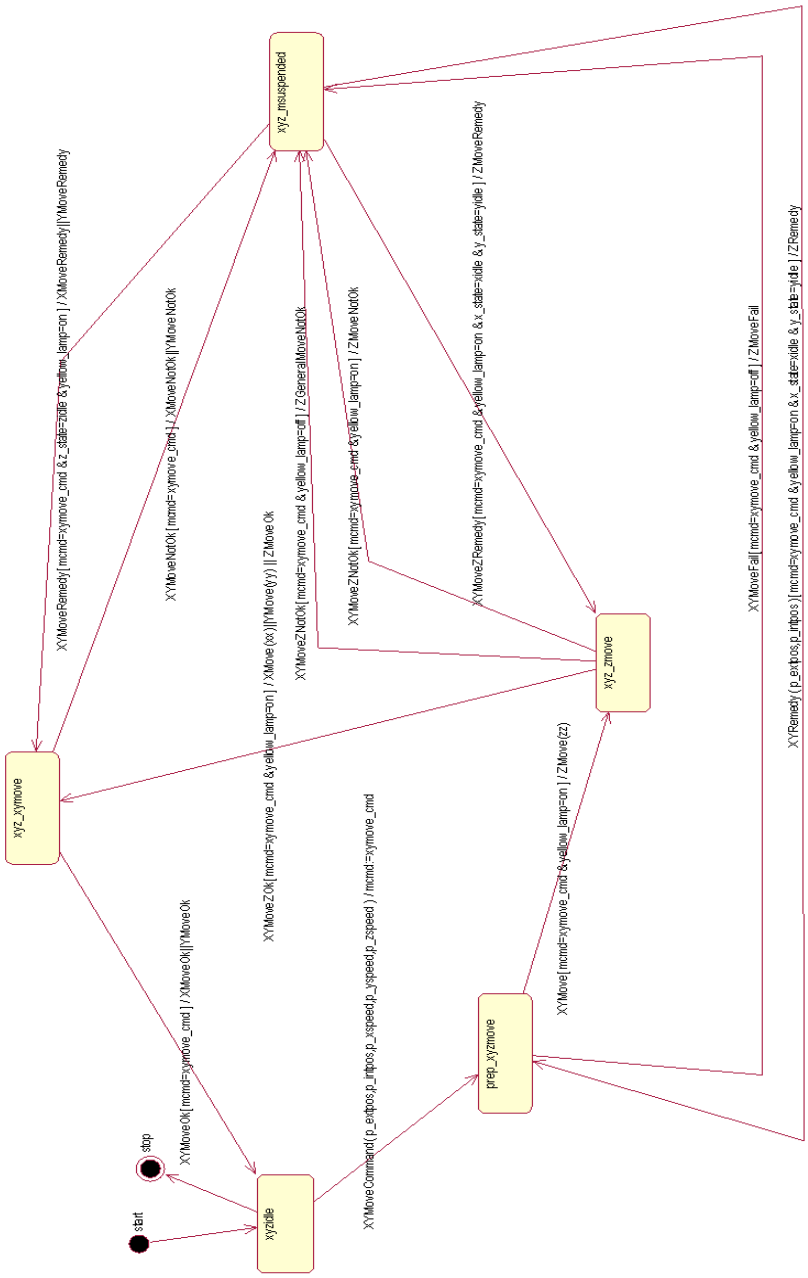


A.3. Third refinement step of the operation aspirate liquid in Dispenser

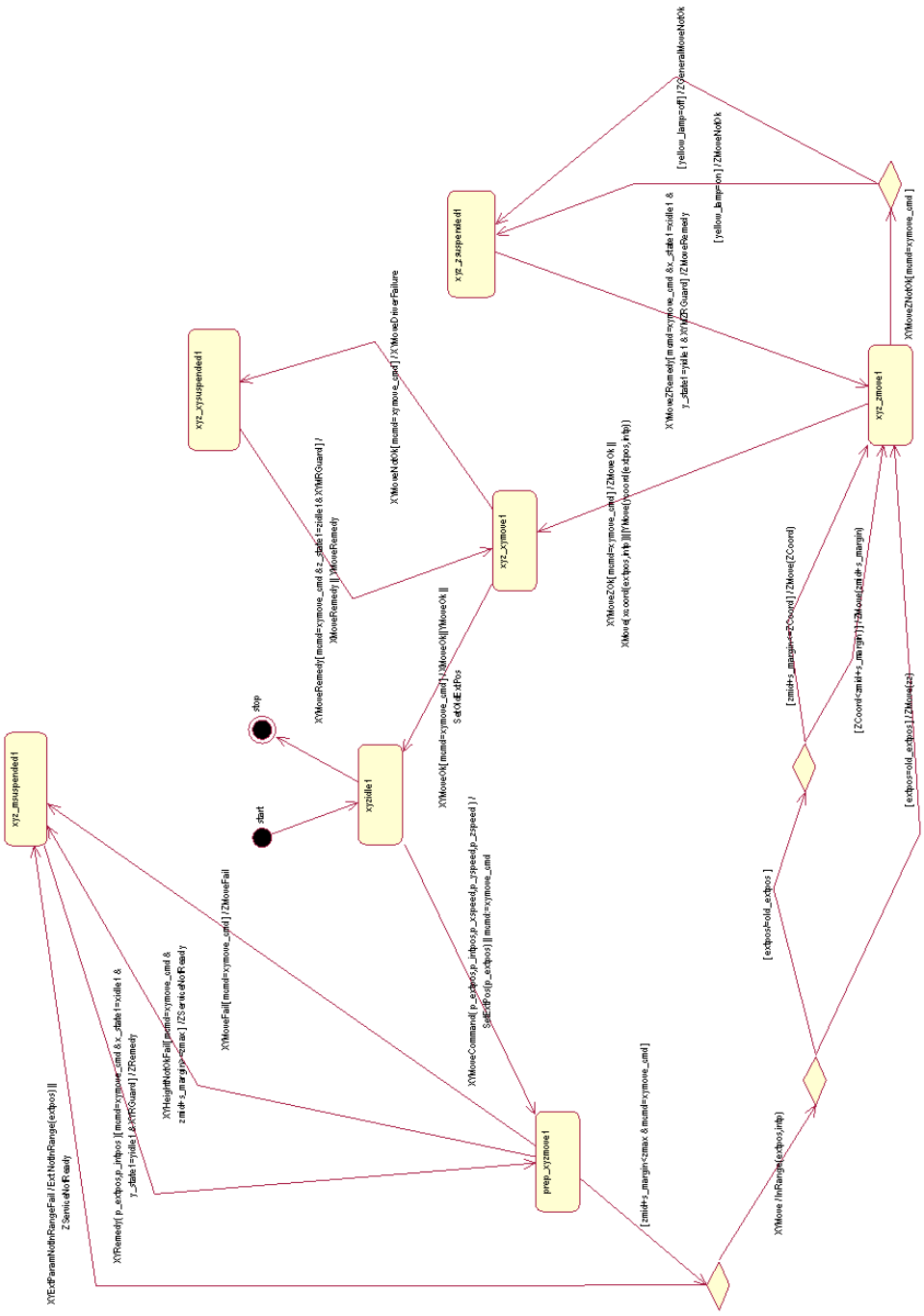


Appendix B

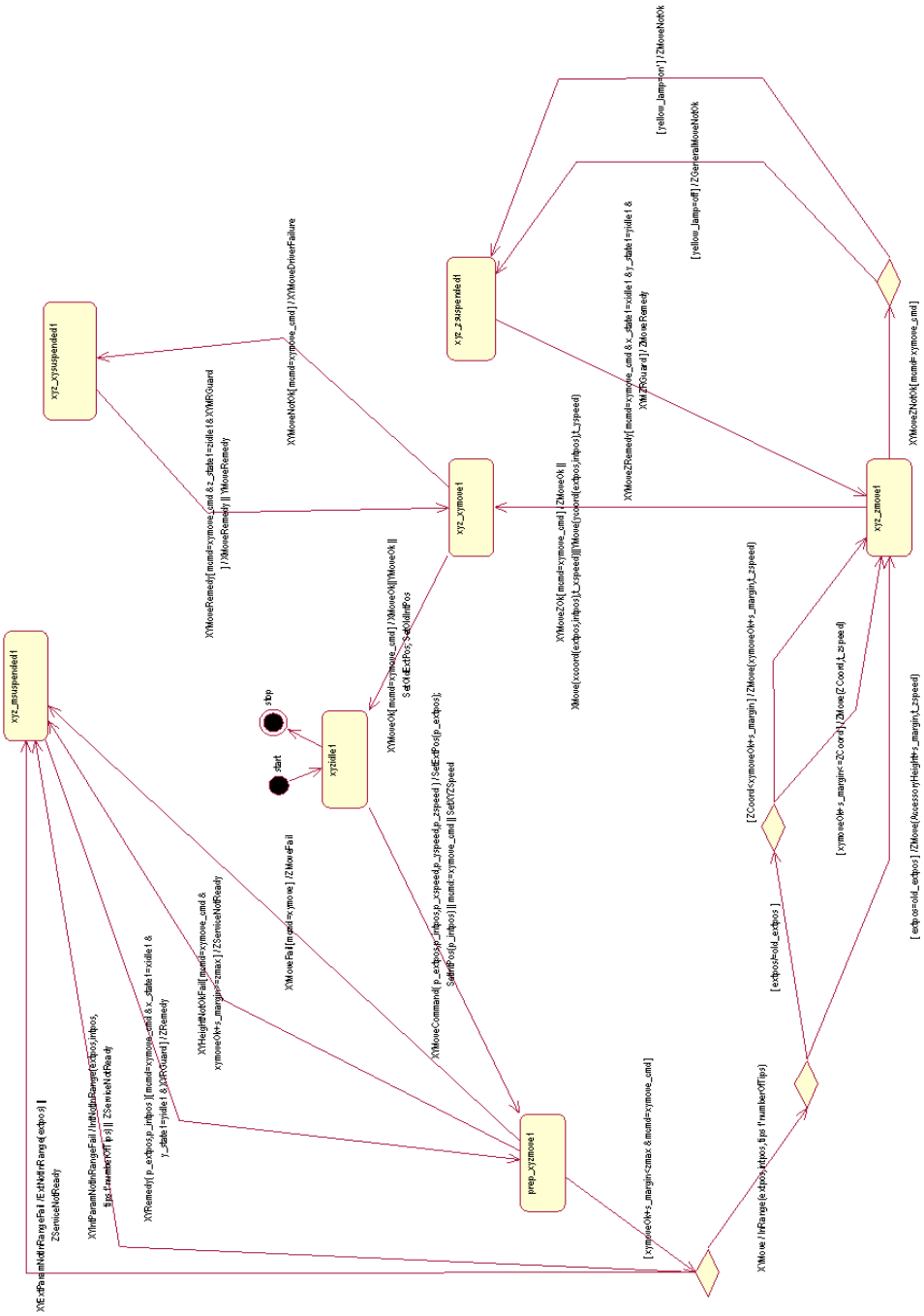
B.1. Specification of the horisontal movement in XYZ-Driver



B.2. First refinement step of the horisontal movement in XYZ-Driver



B.3. Third refinement step of the horisontal movement in XYZ-Driver



Appendix C

C.1. Protocol specification

A protocol consists of three sections: Constants, Configuration and Protocol commands. The language is a subset of the Fillwell command language used at Wallac.

Commands in the constants section:

Begin	Start of section.
XX constname, value	XX is the constant domain, here the x-axis. *constname* is name of constant and *value* is the integer value of the constant
YY constname, value	Same as XX, but the domain is y-axis
ZZ constname, value	Same as XX, but the domain is z-axis
PP constname, value	PP means configuration editable constant. Parameters are the same as for XX.
none constname, list_of_values	Array constant. *constname* is the name of the constant. *list_of_values* is a list of values separated by commas.
End	End of section

Array constants are accessed by writing *constname:index*, where *constname* is the name of the constant and *index* is the name of the loop counter variable.

Configuration verification section:

Begin	Start of section
VerifyConfig pos, AccessoryType	*pos* is the external position of the accessory given as a constant or a value. *AccessoryType* is the type of accessory.
End	End of section.

Protocol commands section:

Begin	Start of section
Protocol commands	Available protocol commands. (See list below).
Loop IndexName, nn	Starts a loop that loops *nn* times. *IndexName*, name of loop counter. This variable can be accessed like a single element constant. Loops can be nested.
End	End of loop or end of section.

Protocol commands:

Command	Parameters	Description
GotoAccessoryPosition	extpos, intpos	Move the dispense head to the plate *extpos* and the internal position *intpos* within that plate.
ZGotoAccessory	zrefpoint, offs	Move the dispense head vertically to the position given by the reference point *zrefpoint* and its offset *offs*.
SetXSpeed	speed	Set the speed for moving in x-direction
SetYSpeed	speed	Set the speed for moving in y-direction
SetZSpeed	speed	Set the speed for moving in z-direction.
Aspirate	amount	Aspirate *amount* of liquid from the current plate
Dispense	amount	Dispense *amount* of liquid into the current plate.
AspiratePrimaryAir	amount	Aspirate *amount* of primary air.
BlowPrimaryAir	amount	Dispense *amount* of primary air.
WashTips	extpos, volume, airVol, repeats, cycle_count, wash_time	Wash the tips in the tip washer. *extpos* is the position of the tip washer, *volume* is the amount of washing liquid to aspirate, *airVol* is the air volume in the tips and *repeats* is the number of dispense-aspirate cycles to go through. *cycle_count* is the number of repeats of cycles to go through including emptying and refilling of the tip washer. *wash_time* is the time the pump-motors should be switched on when filling and emptying the tipwasher.

C.2. Typical course of events for service ReadProtocol

1 The command is read from the line in the protocol pointed on by *line_no*. If line number is too high, then RPF1.
2 Read the first parameter. If the parameter should be a constant name and it is not, then RPF2. If the name should be present in the constant table and it is not, then RPF3.
3 Read the rest of the parameters one by one. If line is too long, then RPF4.
4 Decode and execute the command. Protocol runner checks that it is in the correct state. If it is not, then RPF5.
   4.1. If the command is ***XX***, ***YY***, ***ZZ***, ***PP*** or ***none***, then Protocol runner adds the constants. If the parameters are not valid, then RPF6.
   4.2. If the command is ***Begin***, Protocol runner starts to read constants, to check the configuration of accessories state or to read protocol commands depending on the current protocol section (see Appendix C.1).
   4.3. If command is ***VerifyConfig***, then Protocol runner verifies the accessory configuration. The accessory configuration expected in the protocol should be the same as the actual configuration. If it is not the same, then RPF7
   4.4. If the command is ***Loop***, then Protocol runner can initiate a loop. If not correct number of parameters, then RPF8.
   4.5. If the command is ***GotoAccessory*** or ***ZGotoAccessory***, XYZ-Driver checks parameters and moves. If the number of parameters is not correct, then RPF8. If Dispenser and XYZ-Driver are not idle, then RPF9.
   4.6. If the command is ***Dispense, BlowPrimaryAir, AspiratePrimaryAir*** or ***Aspirate***, the dispense or aspirate function is invoked. If the correct number of parameters was not read from the protocol, then RPF8. If Dispenser and XYZ-Driver are not idle, then RPF9.
   4.7. If the command is ***WashTips***, the TipWashing routine (see Appendix C.3) is started.
   4.8. If the command is ***End***, Protocol runner ends a loop, a section or a protocol.
      4.8.1. If the loop counter is less than the number of times to loop, then jump back to beginning of loop else end the loop.
      4.8.2. If the current line is in the protocol command section but not within a loop, then the command is *end of protocol*.
5 If the command is not *end of protocol*, continue from step 1.

Error reports:
RPF1: Too long protocol.
RPF2: Not a valid constant name.
RPF3: No such constant in protocol.
RPF4: Protocol line too long.
RPF5: Wrong protocol state.
RPF6: Invalid parameters for adding constants.

RPF7: Wrong configuration of accessories
RPF8: Wrong number of parameters for command.
RPF9: Dispenser or XYZ-Driver does not become idle in time.

Remedy for all error reports: Abort execution of protocol and load new protocol.

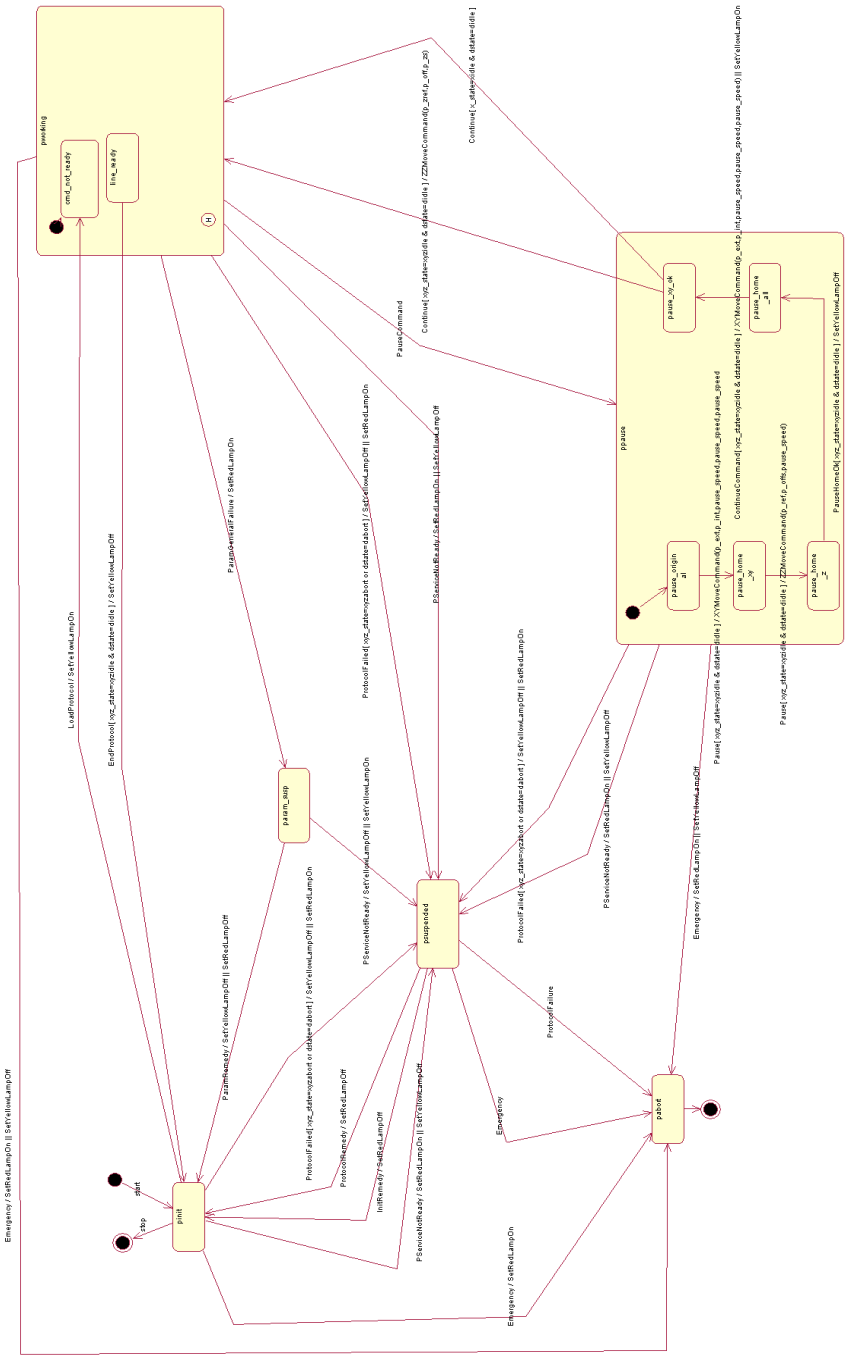
C.3. Typical course of events for service TipWashing

1 Protocol runner checks the parameters. If the parameters are not ok, then TWF1.
2 XYZ-Driver moves the gantry to the position where the tip washer is.
3 If there is liquid in the tips;
  3.1. XYZ-Driver moves down the dispense head into the tip washer and the in-pumps are switched on.
  3.2. Dispenser dispenses the liquid and when the tip washer is full, the in-pumps are switched off. If the pumps are malfunctioning, then TWF2.
  3.3. XYZ-Driver moves up the dispense head above the tip washer and the out-pumps are switched on.
  3.4. When the tip washer is empty, out-pumps are switched off. If the pumps are malfunctioning, then TWF2.
4 Dispenser aspirates primary air.
5 If *cycle_count*>0;
  5.1. The in-pumps are switched on and the variable *cycle_count* is decremented.
  5.2. When the tip washer is full the in-pumps are switched off. If the pumps are malfunctioning, then TWF2.
  5.3. XYZ-Driver moves down the dispense head into the tip washer.
  5.4. Dispenser aspirates and dispenses washing liquid *t_wash_count* number of times
  5.5. If *cycle_count*>0;
    5.5.1. The out-pumps are switched on and XYZ-Driver moves up the dispense head above the tip washer.
    5.5.2. The out-pumps are switched off when the tip washer is empty. If the pumps are malfunctioning, then TWF2.
    5.5.3. Repeat from step 5.
6. Dispenser dispenses primary air and the out-pumps are switched on (*cycle_count=0*).
7. XYZ-Driver moves up the dispense head above the tip washer and when the tip washer is empty the out-pumps are switched off. If the pumps are malfunctioning, then TWF2.
8. Signal that washing is finished.

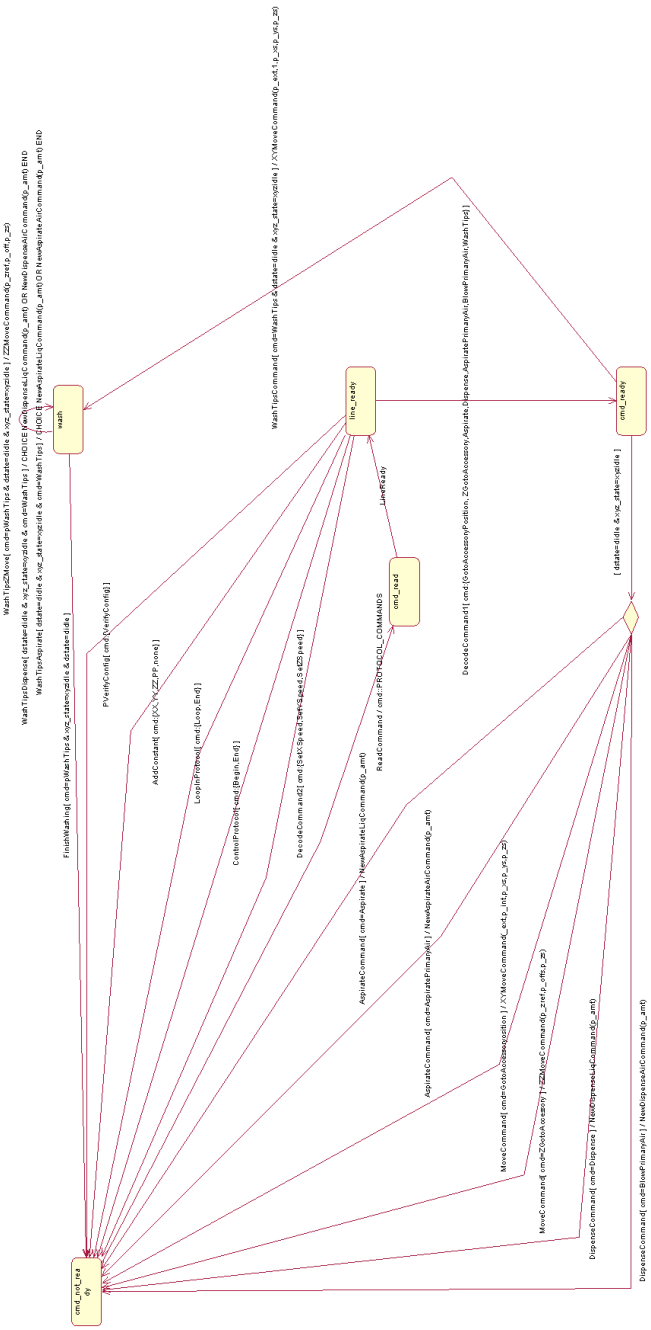
Error reports:
TWF1: Illegal parameters.
TWF2: The pumps are too slow or has stopped working.

Remedy for these errors: Abort execution of protocol and load new protocol.

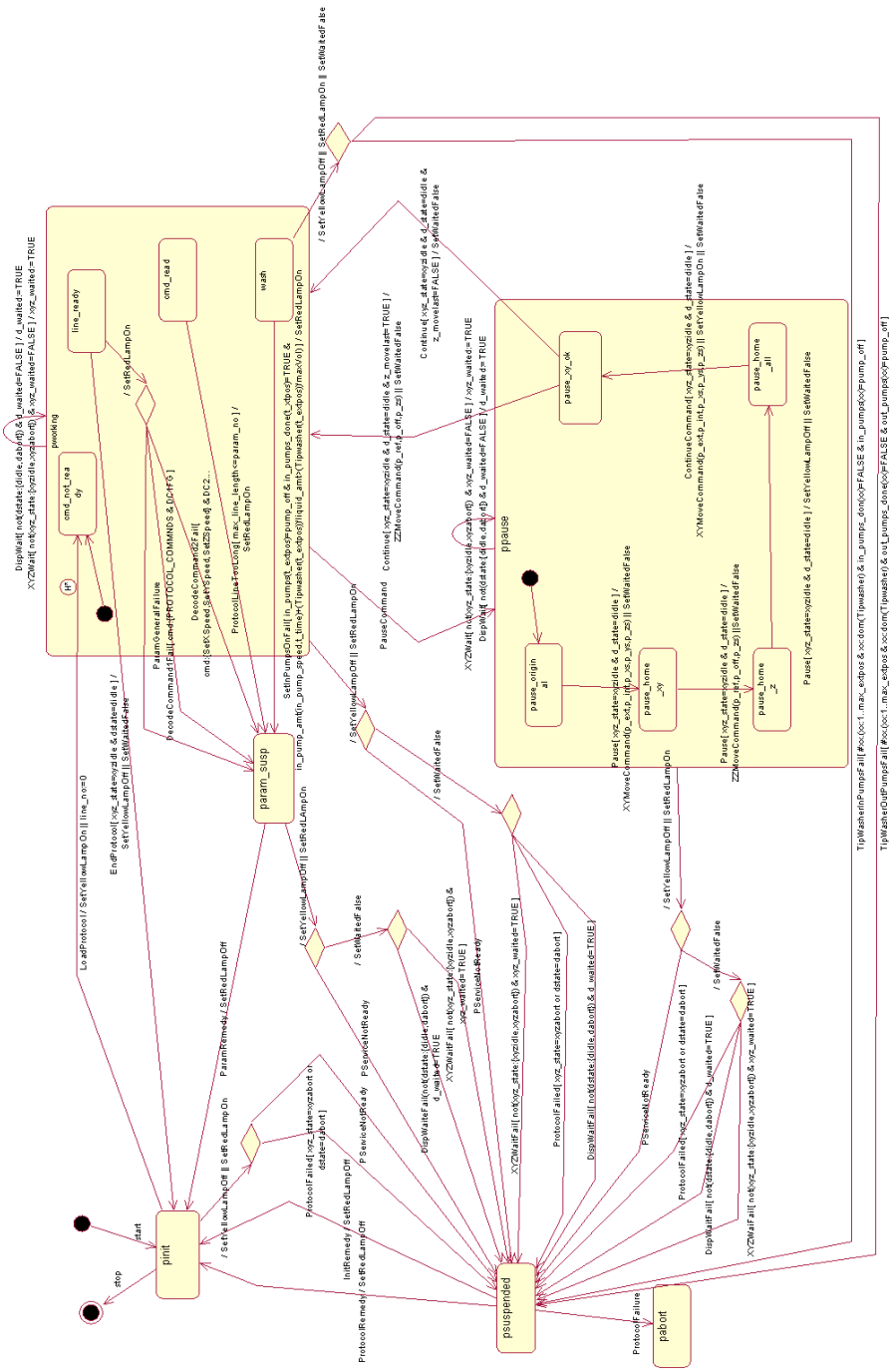
C.4. Specification of the external events in Protocol runner



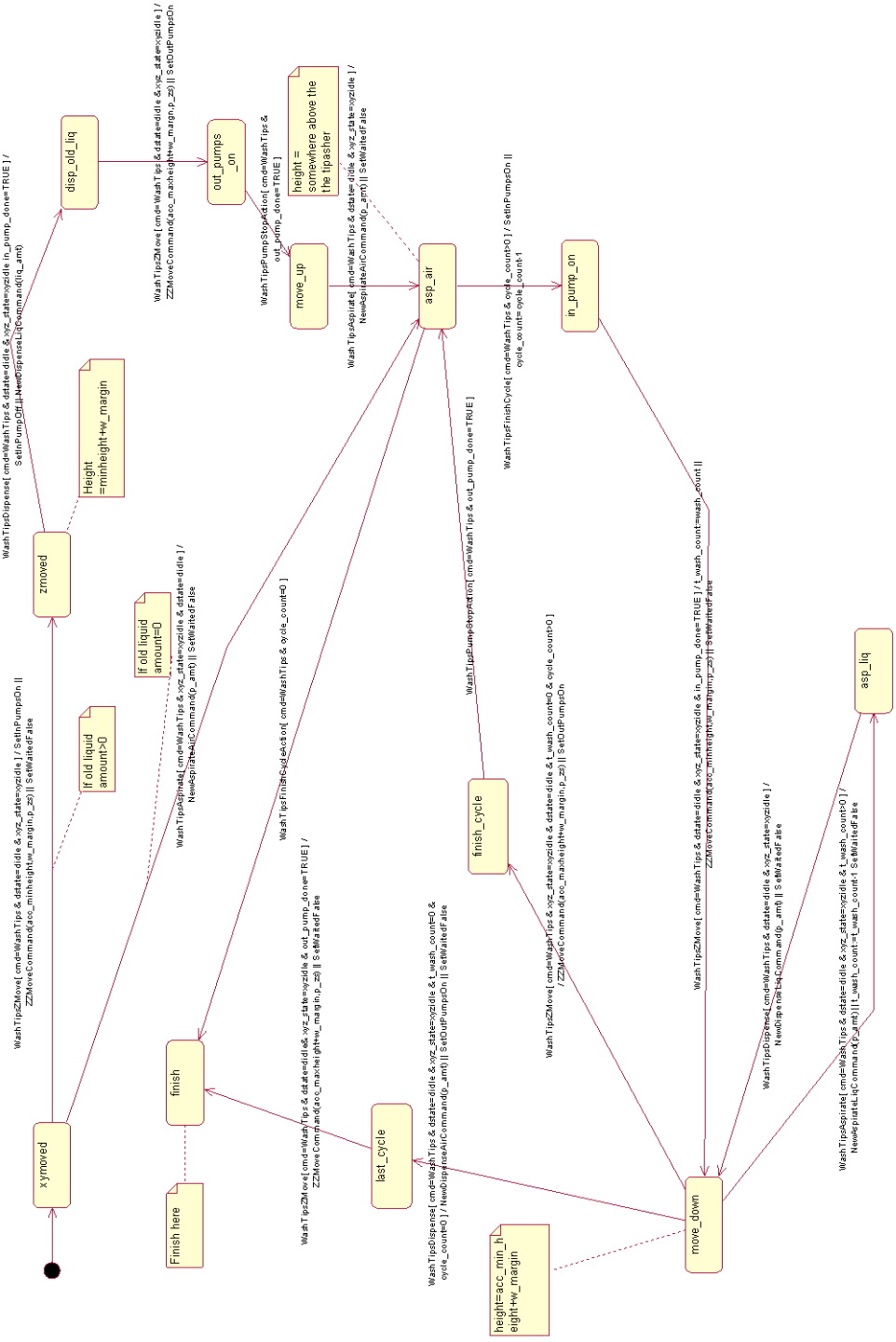
C.5. Specification of the protocol functionality in Protocol runner



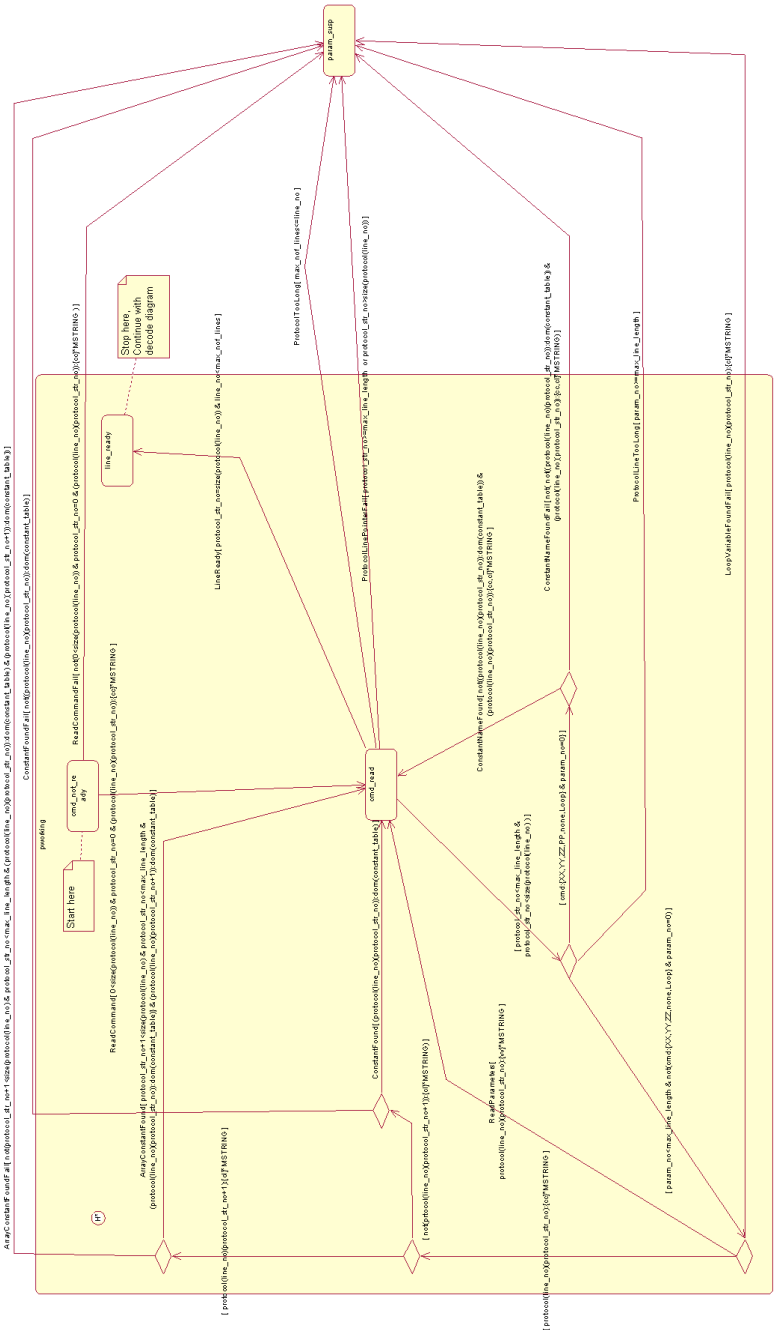
C.6. First refinement step of the external events in Protocol runner



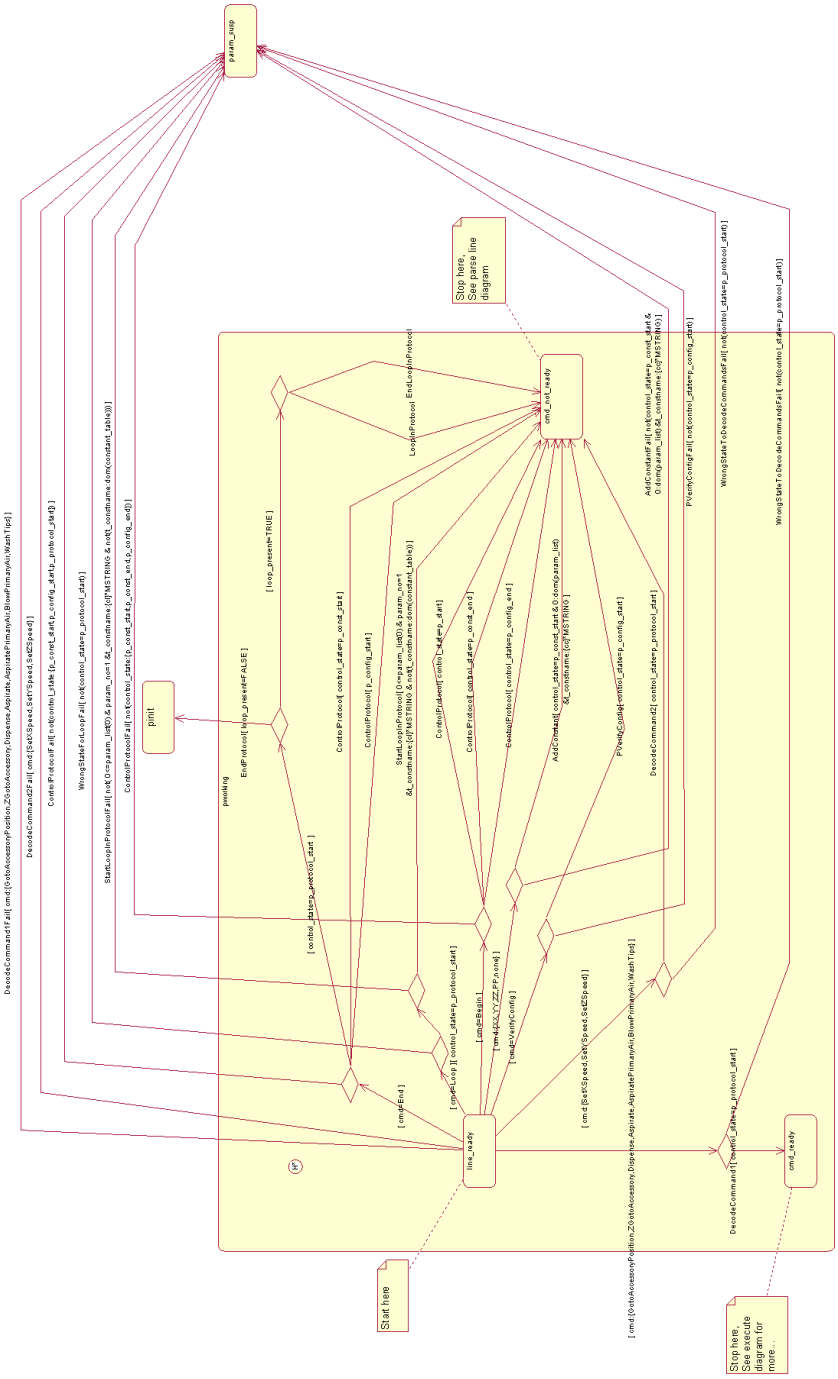
C.7. First refinement step of Tipwasher



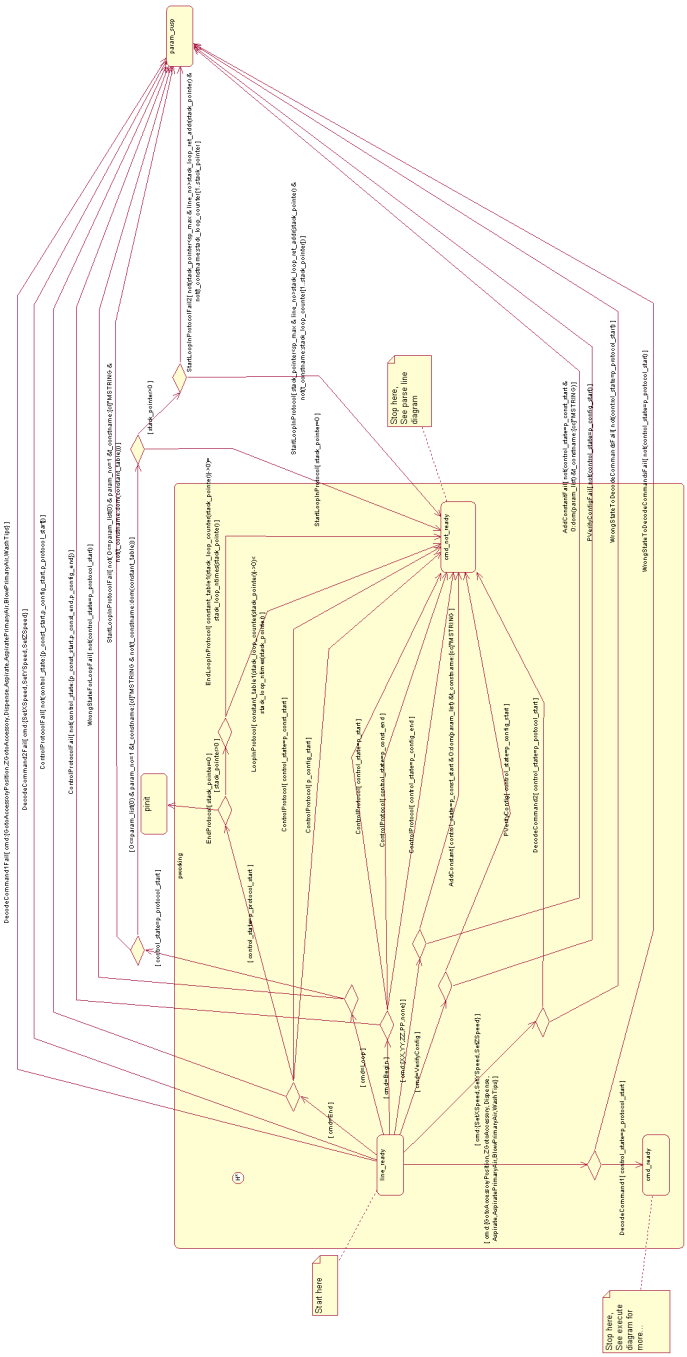
C.8. Second refinement step of line parsing in Protocol runner



C.9. Second refinement step of decoding commands in Protocol runner

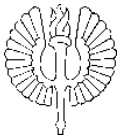


C.10. Third refinement step of decoding commands in Protocol runner



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